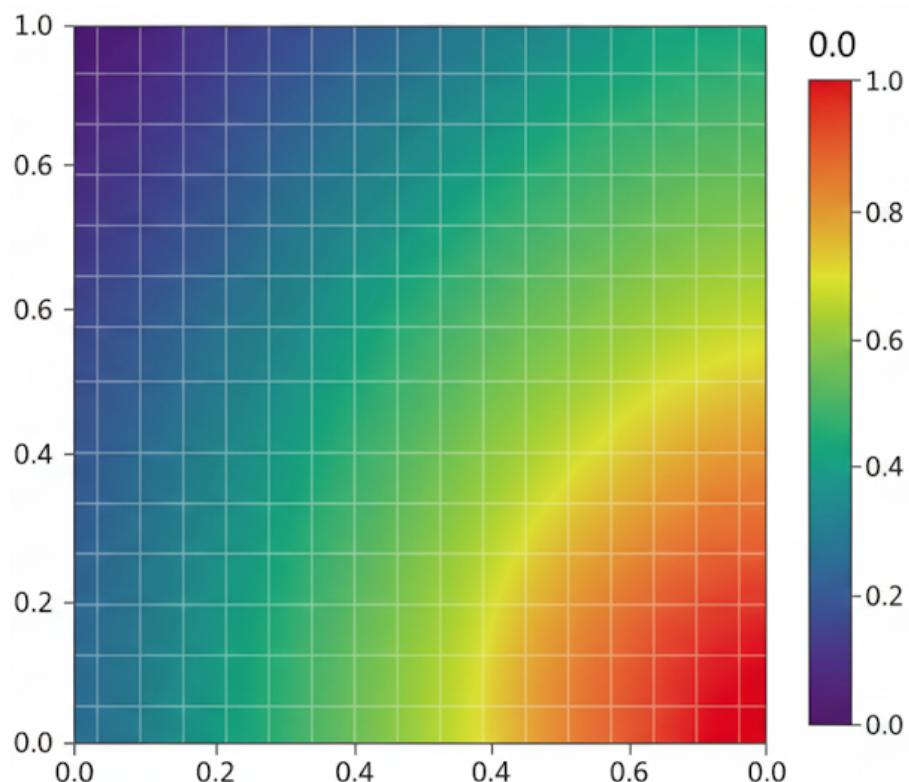


MONITORAMENTO TÉRMICO

SETORIZADO COM FPGA



Isaac Martins de Oliveira Braga e Sousa

Brasília - DF

3 de outubro de 2025

1 Resumo

Este projeto implementa, em uma FPGA Colorlight i9 (Lattice ECP5, clock de $25\tilde{M}Hz$), um sistema de monitoramento térmico setorizado de uma área retangular utilizando 4 sensores DS18B20 posicionados nos vértices (S0–S3). A FPGA executa todo o ciclo do barramento 1-Wire por canal, valida o CRC8 Dallas e converte as leituras para décimos de grau Celsius. Para visualização local e depuração, o sistema dispõe de displays de três dígitos (7 segmentos) capazes de exibir as medições em tempo real.

Após o processamento na FPGA, as amostras são empacotadas com identificadores de origem e transmitidas ao computador por um enlace serial UART (8N1, 115,200 bps). No PC, um script realiza a leitura dos bytes recebidos, reconstrói as temperaturas de S0 a S3 e aplica um algoritmo de interpolação bilinear para produzir um mapa de calor 2D da área monitorada. A visualização emprega uma paleta contínua de cores (frio quente) e pode operar em *modo automático* (normalização min–max por quadro) ou em *modo ancorado* (faixa fixa, com saturação para valores fora dos limites), permitindo comparações consistentes ao longo do tempo.

O resultado é uma *pipeline* completa sensor, FPGA, PC/heatmap, com ênfase em temporizações corretas do 1-Wire, integridade por CRC, tratamento de *timeouts* e empacotamento, possibilitando a identificação de *hotspots* e variações térmicas espaciais e temporais. Como trabalhos futuros, prevê-se a inclusão de limiares de alarme, registro contínuo de dados para análise e reconhecimento de padrões, além da renderização direta de vídeo do mapa de calor na própria FPGA.



Figura 1: Diagrama Resumido do sistema

2 Componentes do Hardware

1. FPGA Colorlight i9
2. 2x Proboard 830 pontos
3. 4x Sensores DS18B20
4. Jumpers para ligações
5. 4x Display de 7 segmentos de 3 dígitos 5631AS, cátodo comum
6. Resitores 4.7 kΩ (*Pull-up* DS18B20)
7. Resistores 330 Ω (Leds dos Displays)
8. Estrutura de suporte simples (impressão 3D) para os sensores
9. Ftdi ft232rl (Conversor Serial/USB)

3 Arquitetura de Alto Nível

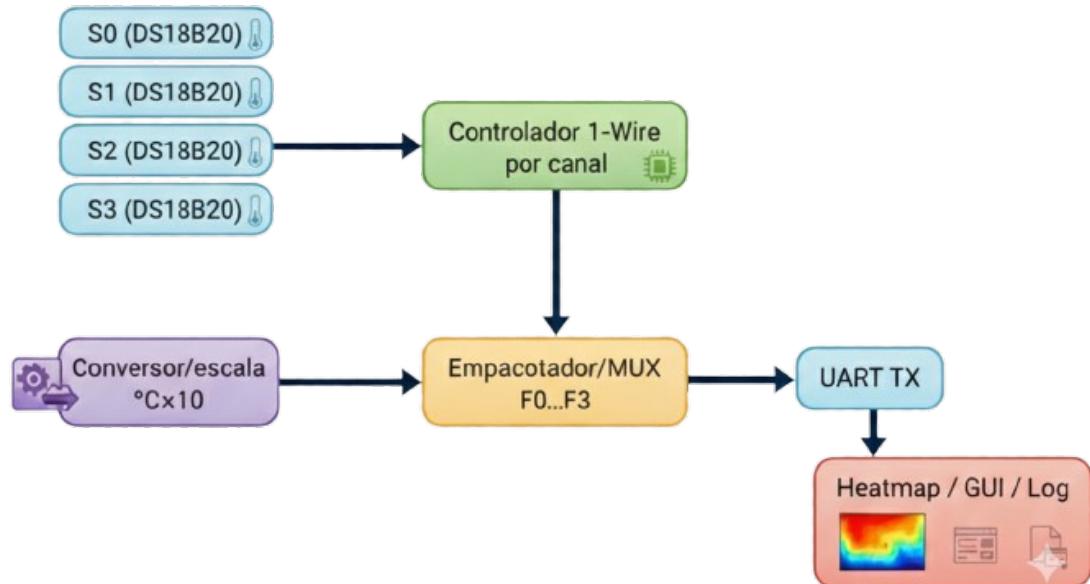


Figura 2: Diagrama alto nível do Sistema

A aquisição começa nos quatro sensores DS18B20 posicionados nos vértices do retângulo monitorado (S0–S3). Cada sensor é alimentado a 3,3 V, compartilha o GND e tem sua linha de dados -DQ- própria com pull-up de $4,7\text{ k}\Omega$ para 3,3 V. Essa linha é do tipo open-drain: o DS18B20 ou a FPGA só “puxam” para zero; o nível alto vem do resistor. Quando a FPGA inicia um ciclo de medição, cada DS18B20 realiza uma conversão com resolução típica de 12 bits (passo de $0,0625\text{ }^{\circ}\text{C}$) e armazena o resultado no scratchpad, junto com um CRC-8 para integridade.

Em cada DQ há um controlador 1-Wire dedicado, implementado como uma máquina de estados temporizada a partir de um tick de 1 μs . O ciclo resumido é: pulso de reset, detecção de presença do sensor, SKIP ROM, CONVERT T, espera de conversão, READ SCRATCHPAD (9 bytes) e checagem de CRC. A leitura respeita as janelas de amostragem do protocolo (write-0/write-1/read-slot), e qualquer anomalia gera timeout e invalida a amostra. Ao final, o bloco entrega a temperatura bruta em dois bytes, em complemento de dois, exatamente como sai do DS18B20.

Logo depois, um conversor/escala transforma esse valor bruto em décimos de grau Celsius. Como o DS18B20 fornece um valor em unidades de $1/16\text{ }^{\circ}\text{C}$, fazemos uma operação inteira equivalente a multiplicar por 10 e dividir por 16 (shift aritmético), preservando sinal e evitando ponto flutuante. Resultado é um inteiro com sinal, por exemplo 253 para $25,3\text{ }^{\circ}\text{C}$.

As quatro leituras convertidas seguem para o empacotador/MUX, que organiza um quadro simples: para cada sensor, envia um byte de endereço (F0, F1, F2, F3) seguido do LSB e do MSB da temperatura (little-endian). Um ciclo completo ocupa 12 bytes: F0 TL TH F1 TL TH F2 TL TH F3 TL TH. Se algum canal estiver inválido (CRC falhou ou timeout), pode-se transmitir um sentinela (p. ex., 0x8000) para o parser do PC tratar adequadamente. Por exemplo, se em uma leitura tivermos: F0 DF 00 F1 1D 01 F2 3B 01 F3 13 01, então significa que o sensor 0 (F0) enviou 0x00DF, ou seja, 223, ou ainda melhor, $22,3^{\circ}\text{C}$.

Esse fluxo alimenta o UART TX configurado em 115 200 bps, 8N1, nível TTL a 3,3 V. O transmissor mantém a linha em idle alto, gera o start bit baixo, emite os 8 bits LSB-first no intervalo de 8,68 μs por bit e finaliza com o stop alto. O sinal tx segue para o RX do conversor USB-serial (FTDI ft232rl), com terra comum. O handshaking entre o empacotador e o UART garante que cada byte só é colocado na linha quando o transmissor está pronto, evitando sobrecarga.

No computador, um parser leve lê o fluxo serial, reconhece a sequência F0→F3 e reconstrói S0..S3 em décimos de $^{\circ}\text{C}$. Com esses quatro pontos e a convenção geométrica ($S0=(0,0)$, $S1=(0,1)$, $S2=(1,1)$, $S3=(1,0)$), aplica-se interpolação bilinear para preencher uma grade 2D: primeiro interpola-se ao longo do topo e da base (S1-S2 e S0-S3) e, em seguida, entre essas duas linhas para cada y do quadro. O campo escalar resultante é normalizado para cores em dois modos: automático (min–max por quadro, realçando variações instantâneas) ou ancorado (faixa fixa, permitindo comparação histórica e saturando valores fora dos limites). Uma paleta — do azul ao vermelho/branco — produz o mapa de calor, exibido em tempo real.

Os valores também podem ser mostrados em displays de 7 segmentos de 3 dígitos (modelo 5631AS) por multiplexação de dígitos, com resistores de $330\text{ }\Omega$ por segmento; o ponto decimal permite exibir décimos (por exemplo, 25.3). Assim, toda a cadeia — sensores, controle 1-Wire, conversão para $^{\circ}\text{C}\times 10$, empacotamento, UART e renderização por interpolação bilinear — fecha uma pipeline coerente, robusta e fácil de instrumentar do físico ao visual.

Nessa implementação construímos nossos módulos utilizando a linguagem de descrição de hardware (HDL) Verilog, para a construção do mapa de calor setorizado (heatmap) utilizamos algoritmos e bibliotecas da linguagem python.

4 Protocolos de Comunicação

4.1 1-Wire (One-wire)

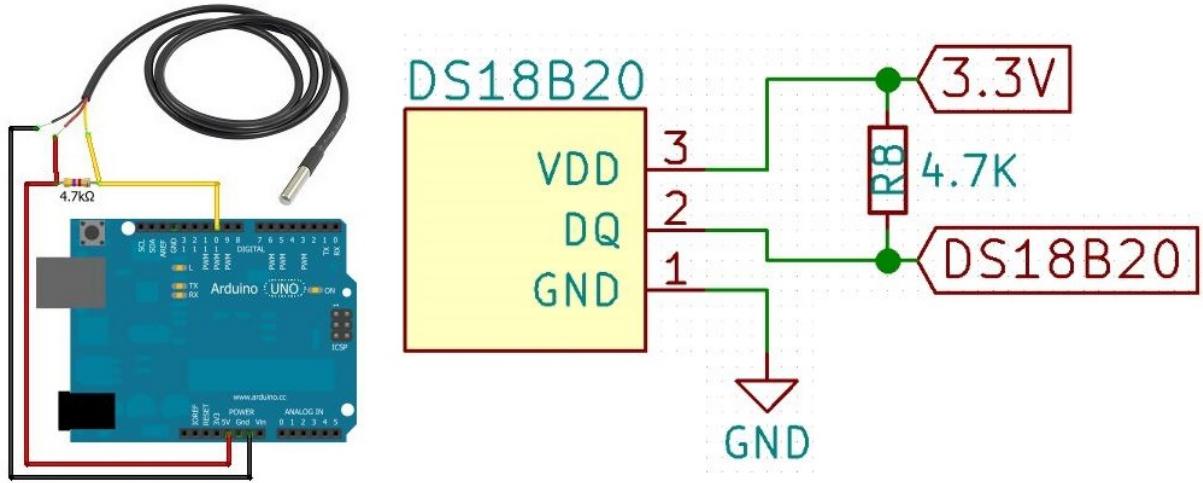


Figura 3: DS18B20, protocolo one-wire

O 1-Wire é um barramento de um único fio de dados (Fio Amarelo) com comportamento open-drain: ninguém coloca “1”ativamente na linha; tanto a FPGA quanto o DS18B20 só puxam para 0 quando precisam. O nível alto vem do pull-up externo (4,7 k Ω a 3,3 V no nosso caso). A linha em repouso fica em alto (idle).

Toda a comunicação é cronometrada pelo master (a FPGA). O diálogo começa com um reset: a FPGA mantém a linha em 0 por 480 μ s e solta. Se houver sensor presente, ele responde com um presence pulse (puxa a linha a 0 por 60–240 μ s). Passado isso, entramos na fase de slots de 1-Wire, cada qual com 60–70 μ s:

- **Escrita (master -> sensor):** para “1”, o master puxa a linha por 6 μ s e solta; para “0”, mantém em 0 por 60 μ s.
- **Leitura (sensor -> master):** o master inicia o slot puxando a linha por 6 μ s e solta; o sensor devolve o bit mantendo 0 (para “0”) ou deixando alto (para “1”). O master amostra tipicamente por volta de 12–15 μ s após o início do slot. O restante do slot fica alto para “recuperar” a linha.

Em nível de comandos, o DS18B20 entende dois grupos, a saber, os comandos da ROM (endereçamento do dispositivo pelo seu código de 64 bits) e os Function commands (converter, ler scratchpad, etc.). Como adotamos um sensor por fio (quatro DQs independentes), usamos sempre SKIP ROM (0xCC) — não precisamos endereçar individualmente. Na sequência teremos então CONVERT T (0x44), após a conversão (tempo depende da resolução), fazemos novo reset+presence e enviamos READ SCRATCHPAD (0xBE) para ler os bytes. O scratchpad traz 9 bytes: Temp_LSB, Temp_MSB, TH, TL,

Config, Rsv \times 3, CRC. Os 12 bits de temperatura vêm em complemento de dois com passo de 1/16 °C (0,0625 °C) — os 4 LSBs são a parte fracionária. A resolução (9–12 bits) define o tempo típico de conversão: 94 ms (9 b), 188 ms (10 b), 375 ms (11 b), 750 ms (12 b). Em nossos módulos - todos os códigos disponíveis nas referências - desenvolvemos o módulo, em verilog, *ds18b20_simple.v*, ele implementa esse processo em três camadas, a saber:

- **Temporização base:** gera um tick de 1 μ s a partir dos 25 MHz, e todos os contadores dos slots/reset são em microssegundos.
- **Camada de bit/slot:** uma micro-FSM garante que cada write-0/write-1/read respeite as janelas; a linha dq é tri-state (dirigida a 0 quando dq_oe=1, solta em Z quando dq_oe=0).
- **Camada de byte/sequência:** outra FSM empacota oito slots em um byte LSB-first, encadeia os bytes dos comandos (0xCC, 0x44, 0xBE) e os dois bytes de temperatura. Ao final, o valor bruto é convertido para décimos de °C com aritmética inteira: como o passo é 1/16 °C, fazemos $(\text{temp_raw} * 10) \gg 4$ (ou “ $\times 10$ e shift à direita de 4”), preservando o sinal.

4.2 Protocolo UART

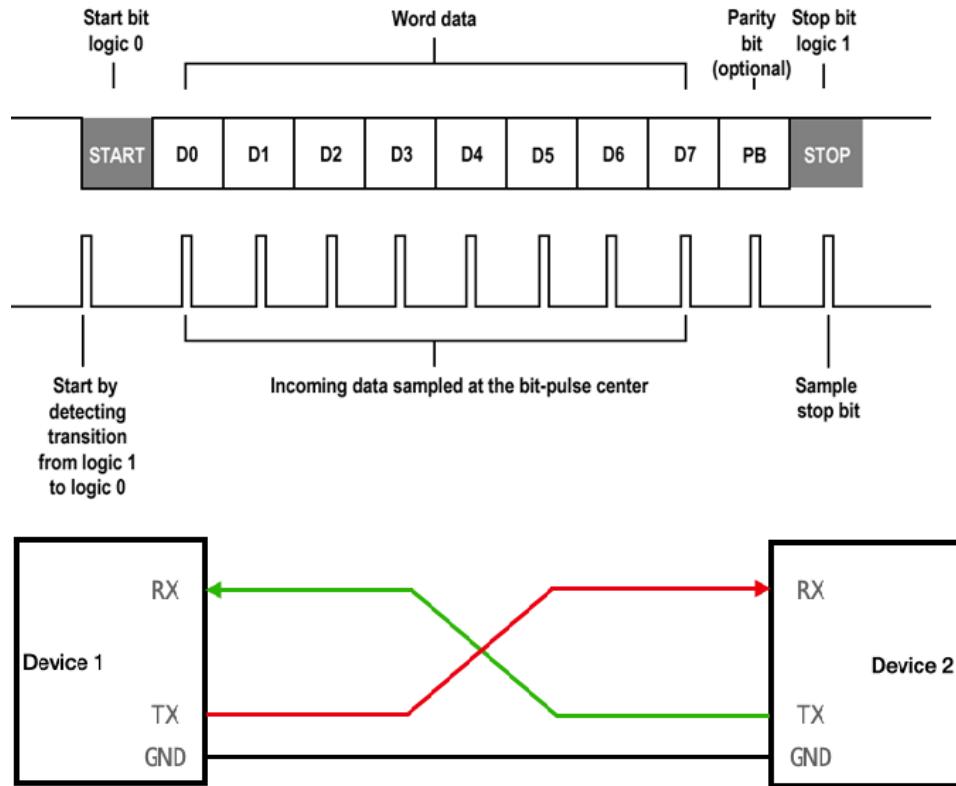


Figura 4: Protocolo UART

O UART (Universal Asynchronous Receiver/Transmitter) codifica bytes em uma linha serial assíncrona mantida em nível alto quando o link está ocioso. Cada caractere é enviado como um frame com início em start bit (nível baixo), seguido de 8 bits de dado transmitidos em ordem LSB-first e finalizado por 1 stop bit (alto). No nosso padrão 8N1, não há bit de paridade e há um único stop. Como o link é assíncrono, emissor e receptor se alinham apenas no flanco do start; daí a importância de um temporizador de baud estável no transmissor e de amostragem precisa no receptor.

Temporização (gerador de baud). Com SYSCLK_HZ = 25 MHz e BAUD = 115200, o tempo de bit é $t_{\text{bit}} \approx \frac{1}{115200} \approx 8,68 \mu\text{s}$. Em clock de 25 MHz (período 40 ns), isso corresponde a 217 ciclos por bit. Usamos um divisor inteiro DIV = 217, produzindo baud $\approx 115207,37$ bps com erro de $\approx +0,006\%$, muito abaixo da tolerância típica de UART (± 1 a $\pm 2\%$). Um caractere 8N1 consome 10 bits (start + 8 + stop), logo $\approx 86,8 \mu\text{s}$ por byte.

O bloco uart_transmitter.v implementa o gerador de baud e a FSM de envio, expondo data_in, start_tx, busy/ready e a saída tx em nível TTL de 3,3 V. Acima dele, temperature_to_uart.v realiza a serialização dos inteiros com sinal em décimos de °C para little-endian (LSB, depois MSB) e insere os bytes de identificação do sensor (F0..F3). O módulo multi_temperature_uart_4sensors.v orquestra o quadro completo com a sequência fixa F0 TL TH F1 TL TH F2 TL TH F3 TL TH, gerando um total de 12 bytes por atualização. Por fim, start_uart_ds18b20.v sincroniza o instante de transmissão com a disponibilidade das leituras válidas dos quatro canais, garantindo que cada quadro reflita um conjunto coerente de medições.

Integração física. A linha tx da FPGA vai ao RX do conversor USB-serial (Ftdi ft232rl) e todos os módulos compartilham o GND. Como o DS18B20 em 12 bits fixa a taxa de atualização em 750 ms.

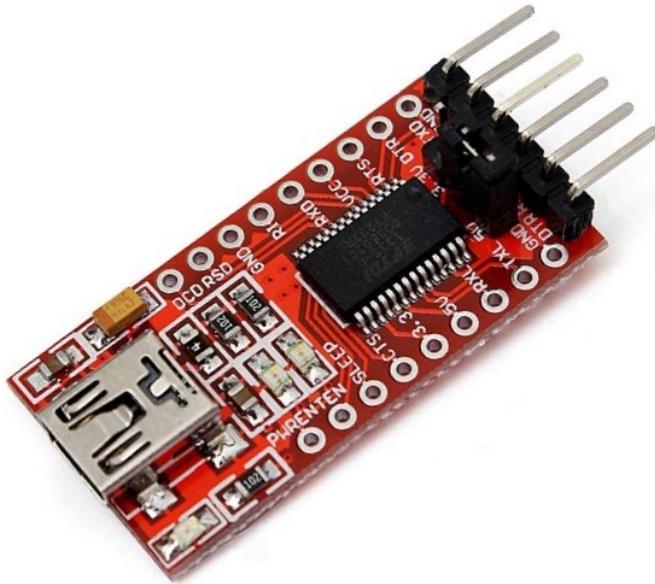


Figura 5: Conversor Serial Ftdi ft232rl

5 RTL - Desenvolvimento dos módulos

Todos os códigos estão disponíveis em nosso github e podem ser acessados livremente pelos links nas referências no final desse documento. Este projeto foi feito utilizando-se da linguagem de descrição de hardware (HDL) Verilog, dessa forma, vamos descrever a implementação dos principais módulos e tecer alguns comentários sobre seus códigos em verilog.

5.1 *ds18b20_simple.v*

```
1 `timescale 1ns/1ps
2 // ds18b20_simple      leitor enxuto do DS18B20 (Verilog-2001)
3 // Entradas : clk (25 MHz), rst_n, dq (1-Wire com pull-up externo
4 // ~4.7k)
5 // Saída      : temperature_x10 (signed, dígitos de C : 253 =>
6 // 25.3 C )
7
8 module ds18b20_simple #(
9     parameter integer SYSCLK_HZ = 25_000_000, // 25 MHz por
10    parameter integer T_CONV_US = 750_000           // tempo m x .
11    parameter integer CYCLES_PER_US = (SYSCLK_HZ/1_000_000);
12
13    input wire clk,
14    input wire rst_n,
15    inout wire dq,
16    output reg signed [15:0] temperature_x10
17 );
18
19 reg [31:0] us_div;
20 wire us_tick = (us_div == CYCLES_PER_US-1);
21
22 always @(posedge clk or negedge rst_n) begin
23     if (!rst_n) us_div <= 32'd0;
24     else         us_div <= us_tick ? 32'd0 : (us_div + 32'd1);
25 end
26
27 // ===== Open-drain no 1-Wire =====
28 reg dq_oe;                      // 1: for a LOW, 0: solta
29 // (pull-up leva a HIGH)
30 assign dq = dq_oe ? 1'b0 : 1'bz;
31 wire dq_in = dq;
32
33 // ===== Tempos em microssegundos =====
34 localparam integer T_RSTL      = 480;
35 localparam integer T_PRESAMPLE = 70;
36 localparam integer T_SLOT       = 64;
37 localparam integer T_W1L        = 6;
38 localparam integer T_WOL        = 60;
39 localparam integer T_RL         = 6;
40 localparam integer T_R_SAMPLE   = 15;
41
42 // ===== Estados (codifica o simples) =====
43 localparam [4:0]
44     S_IDLE          = 5'd0,
45     S_RESETL        = 5'd1,
```

```

45      S_RESETH_WAIT      = 5'd2,
46      S_PRESENCE_DONE    = 5'd3,
47      S_W_SKIP           = 5'd4,
48      S_W_CONVERT         = 5'd5,
49      S_WAIT_CONV         = 5'd6,
50      S_RESETL2           = 5'd7,
51      S_RESETH2_WAIT     = 5'd8,
52      S_PRESENCE2_DONE   = 5'd9,
53      S_W_SKIP2          = 5'd10,
54      S_W_READSCR        = 5'd11,
55      S_R_TEMPL          = 5'd12,
56      S_R_TEMPH          = 5'd13,
57      S_LATCH             = 5'd14;
58
59      reg [4:0] state;
60
61      // ===== Controles e dados =====
62      reg [31:0] us_cnt;
63      reg [7:0] cur_byte;
64      reg [2:0] bit_idx;
65      reg signed [15:0] temp_raw;
66      reg signed [18:0] mult5; // para multiplicar por 5 e depois
67      >> 3
68
69      // ===== Subm quina de slots =====
70      localparam [1:0]
71          SLOT_IDLE  = 2'd0,
72          SLOT_WRITE = 2'd1,
73          SLOT_READ  = 2'd2;
74
75      reg [1:0] slot_mode;
76      reg [31:0] slot_us;
77      reg       slot_bit;
78      reg       rd_bit;
79
80      // ===== Controle de byte =====
81      reg       byte_busy;
82      reg       byte_is_read;
83      reg [7:0] byte_acc;
84
85      // ----- Tarefas (sintetiz veis) -----
86      task start_write_bit; input b; begin
87          slot_mode <= SLOT_WRITE; slot_bit <= b; slot_us <= 32'd0;
88          dq_oe <= 1'b1;
89      end endtask
90
91      task start_read_bit; begin
92          slot_mode <= SLOT_READ; slot_us <= 32'd0; dq_oe <= 1'b1;
93      end endtask
94
95      task step_slot; begin
96          if (slot_mode == SLOT_WRITE) begin
97              if (slot_us == (slot_bit ? T_W1L : T_WOL)) dq_oe <=
98                  1'b0; // solta
99              if (slot_us >= T_SLOT) begin
100                  slot_mode <= SLOT_IDLE;
101                  dq_oe      <= 1'b0;
102              end
103              slot_us <= slot_us + 32'd1;
104          end
105          else if (slot_mode == SLOT_READ) begin

```

```

103         if (slot_us == T_RL) dq_oe <= 1'b0;           // solta
104         para o escravo
105         if (slot_us == T_R_SAMPLE) rd_bit <= dq_in; // amostra
106         if (slot_us >= T_SLOT) begin
107             slot_mode <= SLOT_IDLE;
108             dq_oe <= 1'b0;
109         end
110         slot_us <= slot_us + 32'd1;
111     end endtask
112
113     task start_write_byte; input [7:0] b; begin
114         byte_acc <= b;
115         bit_idx <= 3'd0;
116         byte_is_read <= 1'b0;
117         byte_busy <= 1'b1;
118         start_write_bit(b[0]);
119     end endtask
120
121     task start_read_byte; begin
122         byte_acc <= 8'd0;
123         bit_idx <= 3'd0;
124         byte_is_read <= 1'b1;
125         byte_busy <= 1'b1;
126         start_read_bit();
127     end endtask
128
129     task step_byte; begin
130         if (slot_mode != SLOT_IDLE) begin
131             step_slot();
132         end else if (byte_busy) begin
133             if (byte_is_read) byte_acc[bit_idx] <= rd_bit;
134             if (bit_idx == 3'd7) begin
135                 byte_busy <= 1'b0;
136             end else begin
137                 bit_idx <= bit_idx + 3'd1;
138                 if (byte_is_read) start_read_bit();
139                 else
140                     start_write_bit(byte_acc[bit_idx+1]);
141             end
142         end
143     end endtask
144
145     // ====== FSM principal ======
146     always @(posedge clk or negedge rst_n) begin
147         if (!rst_n) begin
148             state <= S_IDLE;
149             dq_oe <= 1'b0;
150             us_cnt <= 32'd0;
151             slot_mode <= SLOT_IDLE;
152             slot_us <= 32'd0;
153             byte_busy <= 1'b0;
154             temperature_x10 <= 16'sd0;
155             temp_raw <= 16'sd0;
156             mult5 <= 19'sd0;
157         end else if (us_tick) begin
158             // avan a byte/slot a cada 1 us
159             step_byte();
160
161             case (state)
162             S_IDLE: begin
163                 dq_oe <= 1'b1; // for a reset LOW

```

```

163         us_cnt <= 32'd0;
164         state <= S_RESETL;
165     end
166
167     S_RESETL: begin
168         if (us_cnt >= T_RSTL) begin
169             dq_oe <= 1'b0;           // solta
170             us_cnt <= 32'd0;
171             state <= S_RESETH_WAIT;
172         end else us_cnt <= us_cnt + 32'd1;
173     end
174
175     S_RESETH_WAIT: begin
176         // Aqui poder amos checar presen a em ~70us
177         // (dq_in==0),
178         // mas seguimos adiante ap s ~480us solto.
179         if (us_cnt >= T_RSTL) begin
180             us_cnt <= 32'd0;
181             state <= S_PRESENCE_DONE;
182         end else us_cnt <= us_cnt + 32'd1;
183     end
184
185     S_PRESENCE_DONE: begin
186         start_write_byte(8'hCC); // Skip ROM
187         state <= S_W_SKIP;
188     end
189
190     S_W_SKIP: begin
191         if (!byte_busy) begin
192             start_write_byte(8'h44); // Convert T
193             state <= S_W_CONVERT;
194         end
195     end
196
197     S_W_CONVERT: begin
198         if (!byte_busy) begin
199             us_cnt <= 32'd0;
200             state <= S_WAIT_CONV;
201         end
202     end
203
204     S_WAIT_CONV: begin
205         if (us_cnt >= T_CONV_US) begin
206             dq_oe <= 1'b1;           // novo reset
207             us_cnt <= 32'd0;
208             state <= S_RESETL2;
209         end else us_cnt <= us_cnt + 32'd1;
210     end
211
212     S_RESETL2: begin
213         if (us_cnt >= T_RSTL) begin
214             dq_oe <= 1'b0;
215             us_cnt <= 32'd0;
216             state <= S_RESETH2_WAIT;
217         end else us_cnt <= us_cnt + 32'd1;
218     end
219
220     S_RESETH2_WAIT: begin
221         if (us_cnt >= T_RSTL) begin
222             us_cnt <= 32'd0;
223             state <= S_PRESENCE2_DONE;
224         end else us_cnt <= us_cnt + 32'd1;

```

```

224         end
225
226     S_PRESENCE2_DONE: begin
227         start_write_byte(8'hCC); // Skip ROM
228         state <= S_W_SKIP2;
229     end
230
231     S_W_SKIP2: begin
232         if (!byte_busy) begin
233             start_write_byte(8'hBE); // Read Scratchpad
234             state <= S_W_READSCR;
235         end
236     end
237
238     S_W_READSCR: begin
239         if (!byte_busy) begin
240             start_read_byte();           // LSB
241             state <= S_R_TEMPL;
242         end
243     end
244
245     S_R_TEMPL: begin
246         if (!byte_busy) begin
247             temp_raw[7:0] <= byte_acc;
248             start_read_byte();           // MSB
249             state <= S_R_TEMPH;
250         end
251     end
252
253     S_R_TEMPH: begin
254         if (!byte_busy) begin
255             temp_raw[15:8] <= byte_acc;
256             state <= S_LATCH;
257         end
258     end
259
260     S_LATCH: begin
261         // temp_raw est em 1/16 C (signed).
262         // Converter para 0.1 C      (x * 10) / 16 =
263         // (x*5)>>3
264         mult5 <= temp_raw * 5;          // signed * 5
265         temperature_x10 <= mult5 >>> 3;    // divis o
266         // aritm tica por 8
267         state <= S_ILE;                // recome a
268         // ciclo
269     end
270
271     default: state <= S_IDLE;
272 endcase
273 end
274
275 endmodule

```

5.1.1 Panorama

Este módulo encapsula, em um único bloco RTL, todo o ciclo de medição de um DS18B20 em barramento 1-Wire dedicado: reset/presence, disparo da conversão, espera controlada e leitura dos dois bytes de temperatura, já entregando o resultado em décimos de °C (inteiro com sinal). Ele foi pensado para ser autônomo (roda em loop contínuo), com

interface mínima: clk=25 MHz, rst_n, dq (inout) e a saída temperature_x10.

5.1.2 Parâmetros e portas

Os parâmetros SYSCLK_HZ (padrão 25 MHz) e T_CONV_US (padrão 750 000 µs) definem a temporização global. As portas seguem o hardware: dq é um pino tri-state com pull-up externo (4,7 kΩ), e a saída é signed [15:0], onde, por exemplo, 253 corresponde a 25,3 °C.

5.1.3 Tempos característicos do 1-Wire

As constantes internas (em µs) reproduzem as janelas do DS18B20: T_RSTL = 480 para o reset LOW do master, T_PRESAMPLE = 70 como referência clássica de amostragem de presença, T_SLOT = 64 como duração típica de um slot, além dos pulsos curtos de write-1 e read (T_W1L = 6, T_RL = 6) e o LOW prolongado de write-0 (T_W0L = 60). A amostragem de leitura ocorre em T_R_SAMPLE = 15, dentro do slot, quando o escravo já colocou o bit na linha.

5.1.4 FSM principal

1. **Reset/Presence:** força LOW por T_RSTL, solta, aguarda a janela de presença. (Nesta versão, a presença não é explicitamente testada, mas o tempo é respeitado.)
2. **Comandos de disparo:** envia SKIP ROM (0xCC) e CONVERT T (0x44) por meio do motor de bytes.
3. **Espera de conversão:** temporiza T_CONV_US (típico 750 ms 12 bits). Não há polling do bit de busy nesta versão.
4. **Novo reset e comandos de leitura:** SKIP ROM (0xCC) e READ SCRATCH-PAD (0xBE).
5. **Leitura da temperatura:** lê LSB e depois MSB; os 16 bits são armazenados em temp_raw com sinal (formato nativo do DS18B20: 1/16 °C, complemento de dois).
6. **Latching e conversão de escala:** multiplica temp_raw por 5 e faz um shift aritmético à direita de 3 ($\gg 3$), implementando $(x * 10)/16$ com preservação de sinal. O resultado em décimos de °C é colocado em temperature_x10 e o ciclo recomeça em S_IDLE.

Em síntese, ds18b20_simple isola toda a complexidade temporal do 1-Wire em dois motores (slot/byte) e uma FSM clara, entregando um valor já escalado e pronto para empacotamento e transmissão. É compacto e serve como base sólida para as variações com CRC e diagnósticos estendidos.

5.2 *uart_transmitter.v*

```

1 `timescale 1ns/1ps
2
3 module uart_transmitter(
4     // ENTRADAS
5     input wire          clk,
6     input wire          start,      // ideal: pulso de 1 ciclo
7     input wire [7:0]    data,
8     input wire          reset_n,   // reset sincrono, ativo-BAIXO
9
10    // SAIDAS
11    output wire         active,
12    output wire         done,
13    output reg          tx
14);
15
16 parameter integer BAUD           = 115_200;
17 parameter integer FREQUENCIA_FPGA = 25_000_000; // 25 MHz
18
19 // CLKS POR BIT (arredondado p/ inteiro mais proximo)
20 localparam integer CLKS_PER_BIT = (FREQUENCIA_FPGA +
21                                     (BAUD/2)) / BAUD;
22
23 // Largura do contador (minimo 1)
24 localparam integer tamanho_minimo = (CLKS_PER_BIT <= 1) ? 1 :
25                                     $clog2(CLKS_PER_BIT);
26
27 // REGISTRADORES
28 reg [tamanho_minimo-1:0] reg_clock_count =
29     {tamanho_minimo{1'b0}};
30 reg [7:0]                  reg_data          = 8'd0;
31 reg [2:0]                  bit_idx          = 3'd0;
32 reg                      reg_active       = 1'b0;
33 reg                      reg_done         = 1'b0;
34
35 assign active = reg_active;
36 assign done   = reg_done;
37
38 // FSM
39 reg [1:0] state = 2'b00;
40 localparam IDLE      = 2'b00;
41 localparam START     = 2'b01;
42 localparam DATA_BITS = 2'b10;
43 localparam STOP      = 2'b11;
44
45 // (Opcional) Detector de borda de subida para 'start'
46 reg start_d = 1'b0;
47 wire start_rise = start & ~start_d;
48
49 always @ (posedge clk) begin
50     if (!reset_n) begin
51         tx          <= 1'b1;      // linha idle = alta
52         reg_clock_count <= {tamanho_minimo{1'b0}};
53         reg_data        <= 8'd0;
54         bit_idx        <= 3'd0;
55         reg_active     <= 1'b0;
56         reg_done        <= 1'b0;
57         state          <= IDLE;
58         start_d        <= 1'b0;
59     end else begin
60         // atualiza histograma do start (p/ edge detector)
61     end
62 end

```

```

58         start_d <= start;
59
60 // 'done'      pulso de 1 ciclo: zera a cada clock
61 reg_done <= 1'b0;
62
63 case (state)
64     // ----- IDLE -----
65     IDLE: begin
66         tx           <= 1'b1;      // n vel de
67             reposo
68         reg_active    <= 1'b0;
69         reg_clock_count <= {tamanho_minimo{1'b0}};
70         bit_idx       <= 3'd0;
71
72         // Inicio da transmissao
73         if (start_rise) begin
74             reg_data     <= data;      // latch do byte
75             reg_active   <= 1'b1;
76             state        <= START;
77         end
78     end
79
80     // ----- START -----
81     START: begin
82         tx <= 1'b0; // start bit = 0
83
84         if (reg_clock_count < CLKS_PER_BIT-1) begin
85             reg_clock_count <= reg_clock_count + 1'b1;
86         end else begin
87             reg_clock_count <= {tamanho_minimo{1'b0}};
88             state          <= DATA_BITS;
89         end
90     end
91
92     // ----- DATA_BITS -----
93     DATA_BITS: begin
94         tx <= reg_data[bit_idx]; // LSB-first
95
96         if (reg_clock_count < CLKS_PER_BIT-1) begin
97             reg_clock_count <= reg_clock_count + 1'b1;
98         end else begin
99             reg_clock_count <= {tamanho_minimo{1'b0}};
100            if (bit_idx < 3'd7) begin
101                bit_idx <= bit_idx + 1'b1;
102            end else begin
103                bit_idx <= 3'd0;
104                state    <= STOP;
105            end
106        end
107    end
108
109    // ----- STOP -----
110    STOP: begin
111        tx <= 1'b1; // stop bit = 1
112
113        if (reg_clock_count < CLKS_PER_BIT-1) begin
114            reg_clock_count <= reg_clock_count + 1'b1;
115        end else begin
116            reg_clock_count <= {tamanho_minimo{1'b0}};
117            reg_active     <= 1'b0;
118            reg_done       <= 1'b1; // pulso
119            state          <= IDLE;

```

```

119           end
120       end
121
122       default: state <= IDLE;
123   endcase
124 end
125 end
126 endmodule

```

Este módulo implementa o transmissor UART puro (apenas TX) que converte bytes paralelos em um frame 8N1 sobre uma única linha serial (tx). A interface é enxuta: start (idealmente um pulso de 1 ciclo) informa que há um novo byte válido em data[7:0]; a saída tx em nível TTL 3,3 V transmite o start bit (0), os 8 bits de dados em ordem LSB-first e o stop bit (1). Duas flags dão visibilidade ao fluxo: active indica "ocupado" durante a emissão de um caractere, e done pulsa por um ciclo justo no término do stop bit, servindo de gatilho para lógica superior encadear bytes.

A temporização nasce do parâmetro FREQUENCIA_FPGA (25 MHz) e do BAUD (115200). O cálculo CLKS_PER_BIT = (FREQUENCIA_FPGA + BAUD/2) / BAUD arredonda para o inteiro mais próximo, produzindo, para 25 MHz, 217 clocks por bit (8,68 μ s). O contador de bit reg_clock_count tem largura mínima adaptativa via \$clog2(CLKS_PER_BIT); isso evita desperdício de flip-flops e funciona para qualquer baud suportado. Com 25 MHz e 115200 bps, o erro de baud é +0,006%, muito abaixo da tolerância típica ($\pm 1\dots \pm 2\%$).

A lógica é conduzida por uma FSM de quatro estados: IDLE \rightarrow START \rightarrow DATA_BITS \rightarrow STOP \rightarrow IDLE. Em IDLE, tx permanece alto (repouso) e, no flanco de subida de start (detectado por start_rise), o byte é "travado" em reg_data para imunidade contra mudanças durante a transmissão. Em START, a linha vai a 0 por exatamente CLKS_PER_BIT clocks. Em DATA_BITS, o índice bit_idx (0...7) seleciona o LSB primeiro de reg_data, mantendo cada bit por CLKS_PER_BIT clocks. Em STOP, o nível é retomado a 1 pelo mesmo período; ao final, active cai e done pulsa por um ciclo, retornando a IDLE.

Alguns detalhes sutis reforçam a robustez: (i) reset_n é síncrono (apesar do sufixo n), inicializando registradores e colocando tx=1; (ii) done é explicitamente pulsado por um único ciclo, evitando alongamentos por lógica combinacional; (iii) a captura do byte antes do START garante consistência mesmo se a origem mudar data logo após start. Caso se deseje alimentar o transmissor em back-to-back sem lacunas, basta acionar o próximo start no ciclo imediatamente após o done.

5.3 multi_temperature_uart_4sensors.v

```

1 `timescale 1ns/1ps
2
3 // Envia em um nico TX os dados de 4 sensores, por start (ex.:
4 // 1 Hz ou 0,25 s):
5 // Frame de 12 bytes no formato:
6 // [FO, LSB0, MSB0, F1, LSB1, MSB1, F2, LSB2, MSB2, F3, LSB3,
7 // MSB3]
8 module multi_temperature_uart_4sensors #(
9     parameter integer FREQUENCIA_FPGA = 25_000_000, // ex.: 25
10    MHz
11    parameter integer BAUD             = 115_200,
12    parameter [7:0] SENSOR0_ADDR      = 8'hF0,
13    parameter [7:0] SENSOR1_ADDR      = 8'hF1,
14    parameter [7:0] SENSOR2_ADDR      = 8'hF2,

```

```

12     parameter [7:0] SENSOR3_ADDR      = 8'hF3
13 )(
14     input wire          clk,
15     input wire          reset_n,        // reset sincrono,
16     input wire          ativo_baixo,
17     input wire          start,          // pulso para disparar o
18     input wire          envio do frame
19     input wire [15:0] temperatura_0, // sensor 0
20     input wire [15:0] temperatura_1, // sensor 1
21     input wire [15:0] temperatura_2, // sensor 2
22     input wire [15:0] temperatura_3, // sensor 3
23     output wire         tx           // UART TX nico
24 );
25 // ----- Timing UART (8N1 = 10 bits) -----
26 localparam integer CLKS_PER_BIT = (FREQUENCIA_FPGA + (BAUD/2))
27     / BAUD;
28 localparam integer FRAME_CLKS    = CLKS_PER_BIT * 10;
29
30 // ----- Edge detect de 'start' -----
31 reg start_d;
32 always @ (posedge clk) begin
33     if (!reset_n) start_d <= 1'b0;
34     else          start_d <= start;
35 end
36 wire start_rise = start & ~start_d;
37
38 // ----- Snapshot das temperaturas -----
39 reg [15:0] snap0, snap1, snap2, snap3;
40 wire [7:0] LSBO = snap0[7:0];
41 wire [7:0] MSBO = snap0[15:8];
42 wire [7:0] LSB1 = snap1[7:0];
43 wire [7:0] MSB1 = snap1[15:8];
44 wire [7:0] LSB2 = snap2[7:0];
45 wire [7:0] MSB2 = snap2[15:8];
46 wire [7:0] LSB3 = snap3[7:0];
47 wire [7:0] MSB3 = snap3[15:8];
48
49 // ----- UART TX (seu transmissor) -----
50 reg [7:0] data_reg   = 8'h00; // byte atual
51 reg          start_reg = 1'b0; // pulso 1 ciclo para TX
52 wire         active_sig;
53 wire         done_sig;
54
55 uart_transmitter #(
56     .BAUD(BAUD),
57     .FREQUENCIA_FPGA(FREQUENCIA_FPGA)
58 ) u_tx (
59     .clk          (clk),
60     .start        (start_reg),
61     .data         (data_reg),
62     .reset_n     (reset_n),
63     .active       (active_sig), // n o usado aqui
64     .done         (done_sig), // n o usado aqui
65     .tx          (tx)
66 );
67
68 // ----- FSM 4 estados -----
69 localparam [1:0]
70     ST_LOAD    = 2'd0,
71     ST_START   = 2'd1,
72     ST_WAIT    = 2'd2,
73     ST_DONE    = 2'd3;

```

```

71
72 // idx_byte: 0..11 => F0,LSB0,MSB0, F1,LSB1,MSB1,
73 // F2,LSB2,MSB2, F3,LSB3,MSB3
74 reg [1:0] state      = ST_LOAD;
75 reg [3:0] idx_byte   = 4'd0;           // 0..11
76 reg [31:0] cnt       = 32'd0;
77
78 always @(posedge clk) begin
79   if (!reset_n) begin
80     state      <= ST_LOAD;
81     idx_byte   <= 4'd0;
82     cnt       <= 32'd0;
83     data_reg   <= 8'h00;
84     start_reg  <= 1'b0;
85     snap0      <= 16'h0000;
86     snap1      <= 16'h0000;
87     snap2      <= 16'h0000;
88     snap3      <= 16'h0000;
89   end else begin
90     // padrao: no inicio de uma nova transmissao (pulso de 1
91     // ciclo em ST_START)
92     start_reg <= 1'b0;
93
94     case (state)
95       // Escolhe o byte a enviar (primeiro espera start_rise;
96       // os demais seguem)
97     ST_LOAD: begin
98       if (idx_byte == 4'd0) begin
99         if (start_rise) begin
100          // tira snapshot de todos no inicio do frame
101          snap0      <= temperatura_0;
102          snap1      <= temperatura_1;
103          snap2      <= temperatura_2;
104          snap3      <= temperatura_3;
105          data_reg   <= SENSOR0_ADDR;        // byte 0
106          state      <= ST_START;
107        end
108      end else begin
109        // bytes subsequentes (1..11)
110        case (idx_byte)
111          4'd1: data_reg <= LSB0;
112          4'd2: data_reg <= MSB0;
113          4'd3: data_reg <= SENSOR1_ADDR;
114          4'd4: data_reg <= LSB1;
115          4'd5: data_reg <= MSB1;
116          4'd6: data_reg <= SENSOR2_ADDR;
117          4'd7: data_reg <= LSB2;
118          4'd8: data_reg <= MSB2;
119          4'd9: data_reg <= SENSOR3_ADDR;
120          4'd10: data_reg <= LSB3;
121          default: data_reg <= MSB3;      // 4'd11
122        endcase
123        state <= ST_START;
124      end
125    end
126
127    // Dispara a transmissao (pulso de 1 ciclo) e zera
128    // contador
129    ST_START: begin
130      start_reg <= 1'b1;
131      cnt       <= 32'd0;
132      state      <= ST_WAIT;

```

```

129      end
130
131      // Espera o tempo fixo de 1 quadro 8N1
132      ST_WAIT: begin
133          if (cnt < FRAME_CLKS-1) begin
134              cnt <= cnt + 1'b1;
135          end else begin
136              if (idx_byte < 4'd11) begin
137                  idx_byte <= idx_byte + 1'b1;
138                  state <= ST_LOAD;
139              end else begin
140                  idx_byte <= 4'd0;           // pronto p/ proximo frame
141                  (proximo start)
142                  state <= ST_DONE;
143              end
144          end
145      end
146
147      // Ociooso ate proximo start_rise
148      ST_DONE: begin
149          if (start_rise)
150              state <= ST_LOAD;
151      end
152      default: state <= ST_LOAD;
153  endcase
154 end
155 end
156 endmodule

```

Este bloco orquestra o envio do quadro completo de 12 bytes que representa as leituras de quatro sensores. A entrada start (pulso de 1 ciclo) inicia um snapshot atônico das temperaturas (snap0..snap3) para que todo o frame reflita um conjunto coerente. Em seguida, uma FSM de quatro estados (ST_LOAD, ST_START, ST_WAIT, ST_DONE) percorre os 12 slots (índice idx_byte=0..11) na ordem: F0, LSB0, MSB0, F1, LSB1, MSB1, F2, LSB2, MSB2, F3, LSB3, MSB3. Cada byte a enviar é colocado em data_reg durante ST_LOAD; ST_START gera um pulso start_reg de um ciclo para o uart_transmitter; ST_WAIT aguarda o tempo exato de um caractere 8N1 (FRAME_CLKS = CLKS_PER_BIT × 10); ao término, ou avança para o próximo byte, ou finaliza retornando a ST_DONE. O próximo quadro só é armado quando chegar um novo start externo.

A sincronização temporal do multi_temperature_uart_4sensors é determinística porque reutiliza o mesmo CLKS_PER_BIT do transmissor ao calcular FRAME_CLKS. Assim, mesmo sem observar os sinais active/done do uart_transmitter, o espaçamento entre bytes permanece preciso e livre de jitter. Essa abordagem simplifica o design e, na prática, produz o mesmo resultado que handshaking, desde que ambos compartilhem BAUD e FREQUENCIA_FPGA.

Outros cuidados visíveis no código: (i) start_rise evita reentradas em meio a um frame; (ii) o snapshot no byte 0 garante que LSB/MSB corresponderão à mesma amostra por sensor; (iii) os endereços de cada sensor (SENSORx_ADDR, padrão F0..F3) são parametrizáveis, o que facilita reutilização ou expansão de protocolo.

6 Mapa de Calor (Heat map)

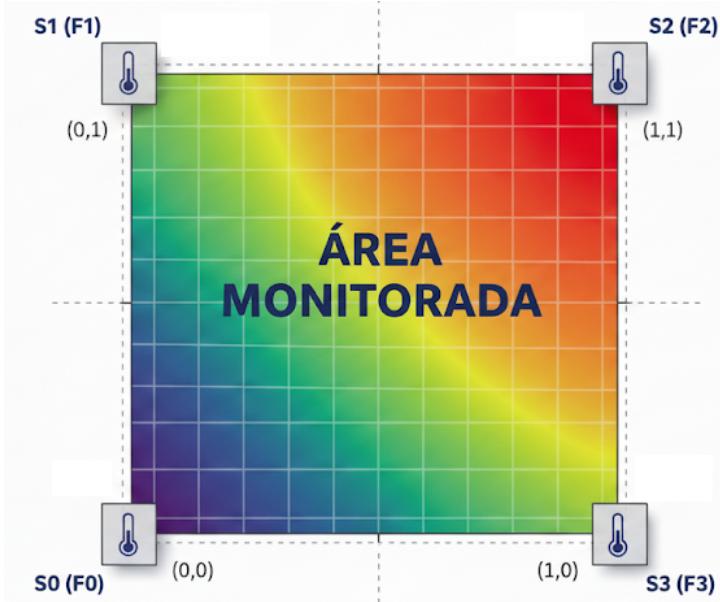


Figura 6: Mapa de calor (Heat Map)

6.1 Funcionamento geral

O mapa de calor traduz quatro amostras discretas ($S_0..S_3$) em um campo escalar 2D contínuo sobre o retângulo monitorado. Adotamos o sistema de coordenadas normalizado $[0,1] \times [0,1]$, fixando $S_0=(0,0)$, $S_1=(0,1)$, $S_2=(1,1)$ e $S_3=(1,0)$. A cada ciclo de atualização, recebemos as temperaturas em °C dos quatro vértices e calculamos, para cada ponto (x,y) do grid, um valor $T(x,y)$. Esse campo é então convertido em cores pela paleta e exibido

6.2 Por que interpolação bilinear?

Com apenas quatro sensores nos cantos, o modelo de melhor compromisso entre simplicidade, estabilidade e custo computacional é a interpolação bilinear. Ela assume que a variação entre os cantos é aproximadamente afim por direção (linear em x, linear em y e linear na combinação de ambos). A bilinear é contínua, determinística (só depende dos quatro cantos), barata (somente somas e multiplicações; em FPGA vira somas e shifts) e monótona em cada direção, evitando artefatos sem dados internos. Na prática, com 4 pontos não há informação para estimar curvaturas internas com segurança; a bilinear torna explícita essa limitação e produz um mapa fiel ao que os sensores suportam inferir.

6.3 Fórmula e implementação espacial

Nos quatro cantos como mostrado na Figura 6 temos um sensor, S_0 , S_1 , S_2 e S_3 , sejam suas teperaturas nesses vértices dadas por:

$$T_{00} = S_0, \quad T_{01} = S_1, \quad T_{11} = S_2, \quad T_{10} = S_3$$

Então, a interpolação bilinear em $(x, y) \in [0, 1]^2$ é dada por:

$$T(x, y) = (1 - x)(1 - y)T_{00} + x(1 - y)T_{10} + (1 - x)yT_{01} + xyT_{11}$$

No script, em python, geramos vetores x e y uniformes (tamanho n), formamos as malhas X,Y e aplicamos a expressão de forma vetorizada para produzir uma grade $n \times n$. Esse grid alimenta a imagem do heatmap (Matplotlib imshow).

6.4 Estrutura do Script

- **Recepção serial e parser:** Uma thread de leitura abre a porta (`-port, -baud=115200`) e executa um parser de estado que reconhece a sequência ID (F0..F3) → LSB → MSB para cada sensor. O valor de 16 bits é interpretado como inteiro com sinal em décimos de °C e convertido para °C (/10.0). O mapeamento de endereços garante que F0..F3 atualizem sempre S0..S3 na ordem correta.
- **Interpolação e rendering:** Quando S0..S3 válidos, chamamos a rotina bilinear para obter a grade $n \times n$ (`-grid`, padrão 80). O resultado é passado a imshow. Os marcadores dos sensores (scatter) e labels de texto (S0: 25.3°C, etc.) ajudam a depurar e a associar visualmente as bordas às cores do mapa.
- **Normalização de cores:** Há dois modos:
 1. **Estático:** (`-fixed VMIN VMAX`): melhor para cenários de alarme/incêndio, pois mantém a escala absoluta (ex.: 15–60 °C), permitindo comparações entre quadros e execuções.
 2. **Dinâmico:** (`min–max` do quadro): realça variações instantâneas. Para evitar flicker, armazenamos `last_vmin/vmax` e só reajustamos a color scale quando a mudança excede um limiar (ex.: 0,05 °C).

7 Testes e Validações

7.1 Objetivos

Formalizar e demonstrar: (i) correção da aquisição 1-Wire (timings e integridade dos dados), (ii) coerência do empacotamento/serialização UART, (iii) resposta do heatmap a aquecimentos localizados, (iv) comportamento comparado entre escala dinâmica (min–max), Procedimento A, e escala fixa, Procedimento B, e (v) utilidade da setorização (identificação de hotspots e persistência).

7.2 Ambiente e Instrumentação

No tópico "Anexos" colocamos as fotos de todo o nosso sistema, que é composto por:

- Bancada com FPGA Colorlight i9
- Estrutura de suporte retangular (Impressão 3D)
- 4× DS18B20 fixados nos vértices do suporte retangular
- 2x Protoboards

- Jumpers para ligações
- Resistores
- Conversor Serial Ftdi ft232rl
- Isqueiro para aquecer os sensores
- Pc com script do Heatmap

7.3 Procedimento A — Escala dinâmica (min–max por quadro)

A paleta reescalada a cada quadro para ocupar todo o range [min,max] instantâneo, realçando contrastes locais.

7.3.1 T0 (Baseline, frio):

Capturamos a imagem inicial. Espera-se um campo quase uniforme (tons frios) com variação suave; S0..S3 próximos entre si.

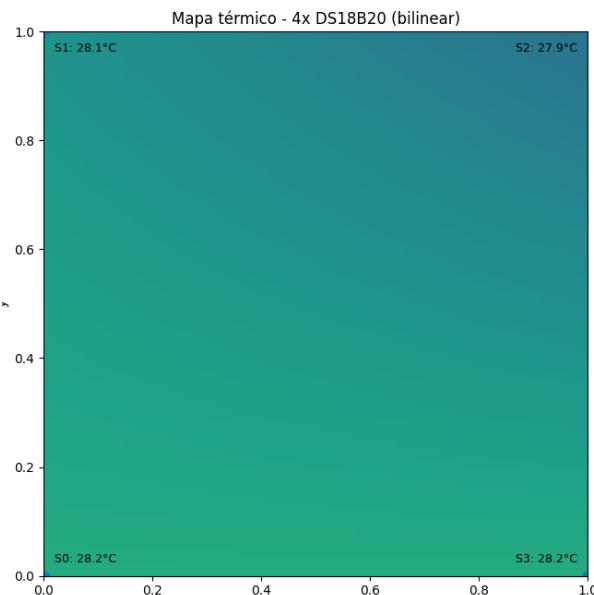


Figura 7: T0

7.3.2 T1 (Aquecimento pontual)

Aplicamos calor em S0 (inferior direito) até uma temperatura bem acima do ambiente, mantendo os demais estáveis. Capturamos a imagem. É esperado que a região próxima a S0 passe a ter tons mais quentes, gradualmente, como pode ser visto nas imagens; Em seguida, Aquecemos também o sensor em S1 de maneira gradual de maneira a ser possível observar o gradiente S0 - S1 com tons mais quentes e as demais regiões em tons mais frios. Em escala dinâmica, o hotspot tende a ocupar a banda vermelho/amarelo, enquanto o restante "desce" para verde/azul—mesmo que o ambiente também aqueça levemente.

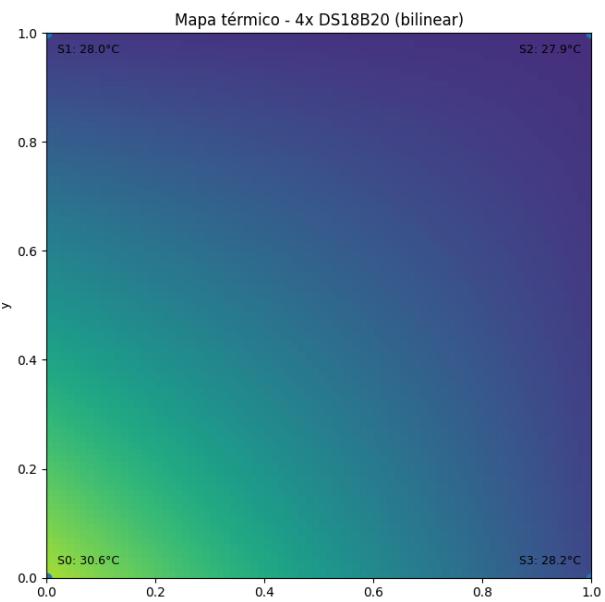


Figura 8: T1: Aquecimento em S0

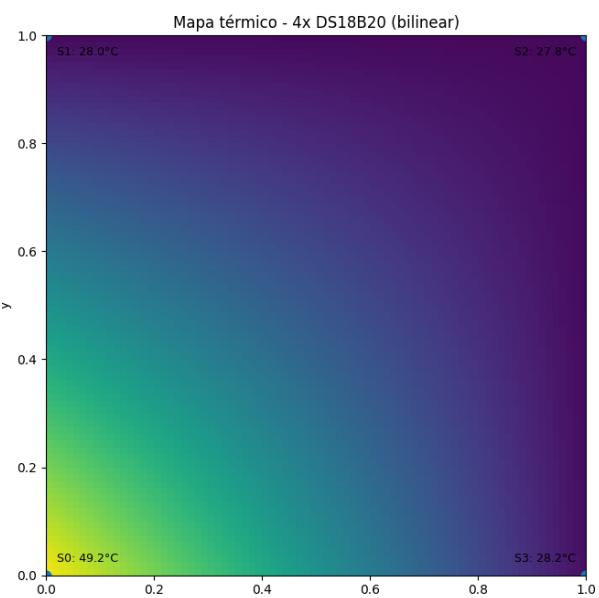


Figura 9: T1: Aquecimento em S0

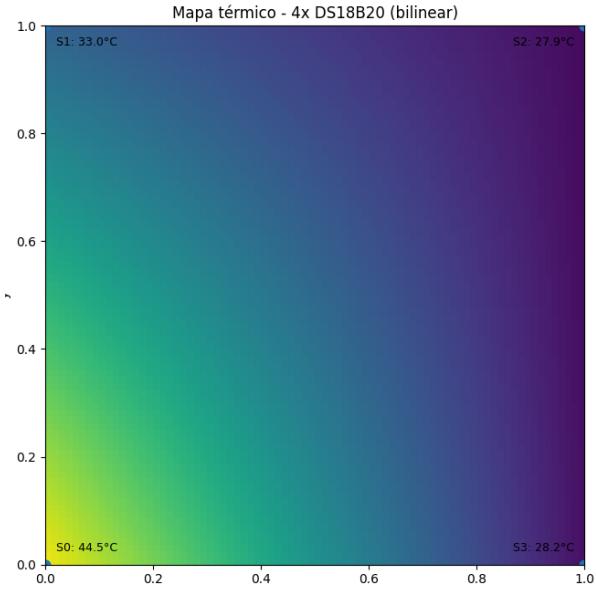


Figura 10: T1: Aquecimento em S1

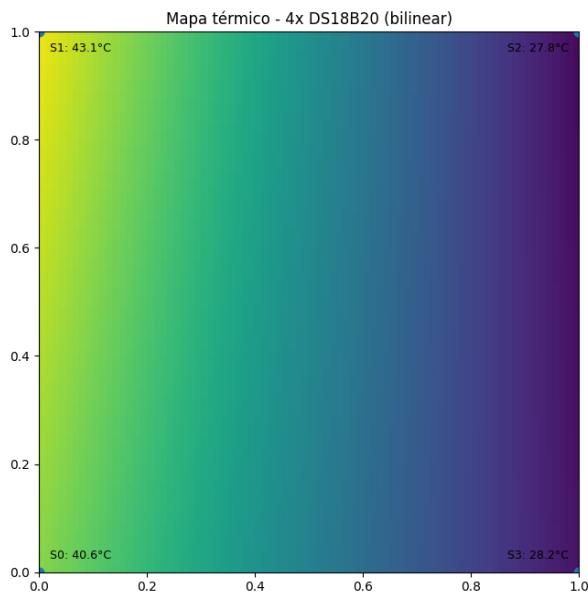


Figura 11: T1: Aquecimento em S1

7.4 T2 (resfriamento)

Retiramos o calor e capturamos a volta gradual às cores frias. Em dinâmica, o vermelho do hotspot desaparece rapidamente porque o range se fecha conforme o máximo cai.

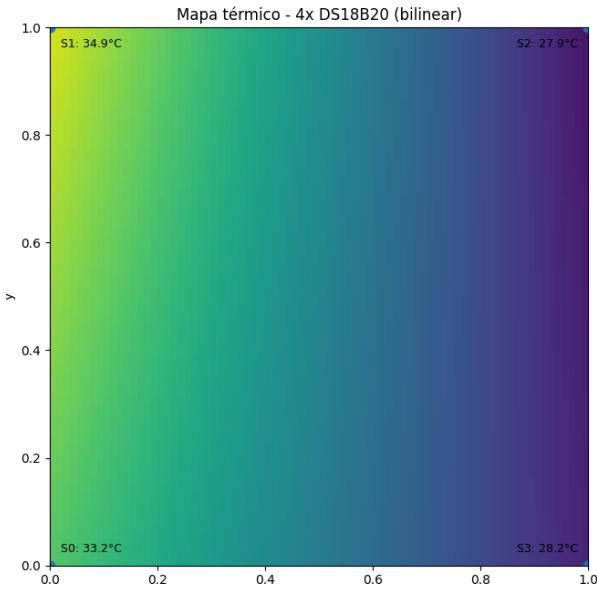


Figura 12: T2: Resfriamento

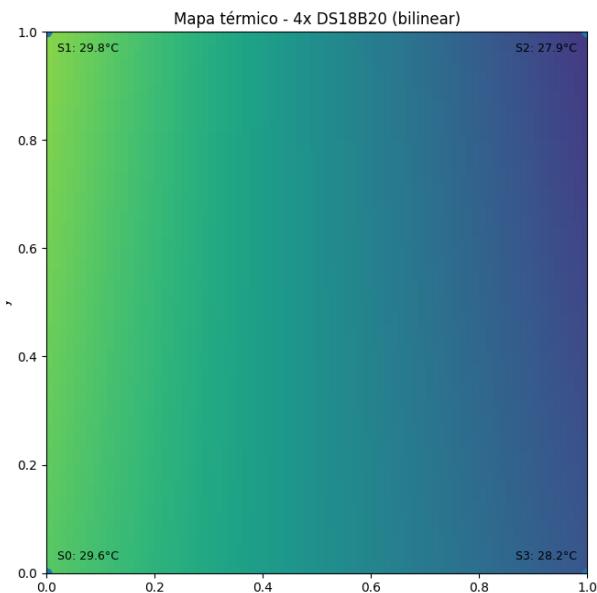


Figura 13: T2: Resfriamento

7.5 Procedimento B — Escala Fixa (25–40 °C)

A paleta permanece anexada, os valores de mínimo e máximo são parametrizáveis, para nosso teste utilizamos mínimo de 25°C e máximo de 40°C, a uma faixa absoluta, permitindo leitura quantitativa e comparação entre quadros e ensaios.

7.5.1 T0 (Baseline, frio):

Com a escala fixa aplicada, capturar a imagem. Espera-se tons azul/ciano (25–30 °C, por exemplo) quase uniformes.

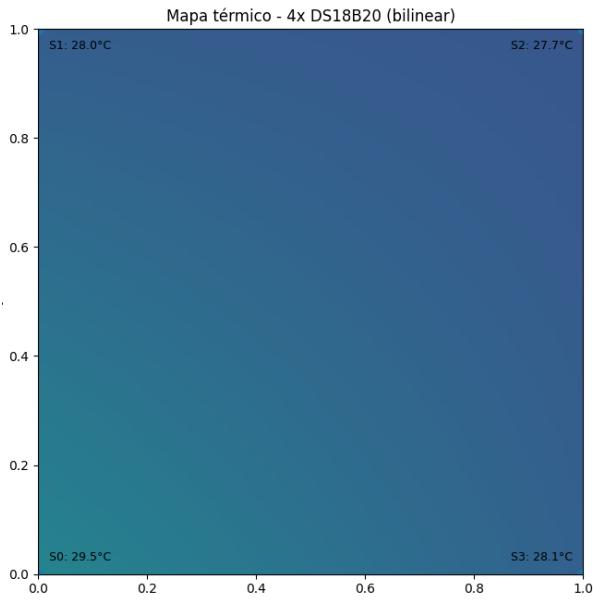


Figura 14: T0

7.5.2 T1 (Aquecimento pontual)

Aplicamos calor no sensor 0. Capturar imagens em +30 s e +60 s. Esperado: cores em torno de S0 migram azul → verde → amarelo → laranja conforme o valor local cruza patamares fixos; setores adjacentes acompanham com gradientes mais suaves. Aplicamos calor em S0 (inferior direito) até uma temperatura acima do ambiente.

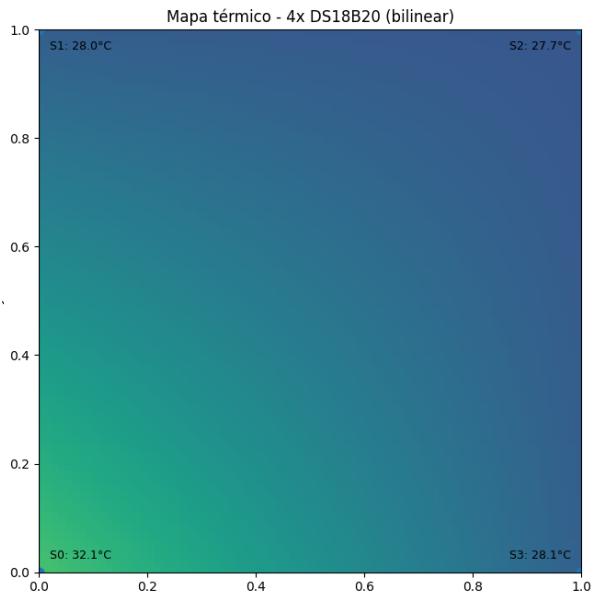


Figura 15: T1: Aquecimento em S0

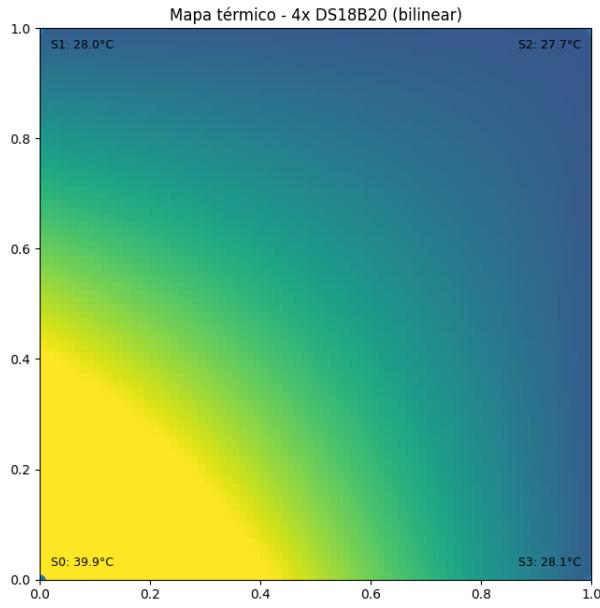


Figura 16: T1: Aquecimento em S0

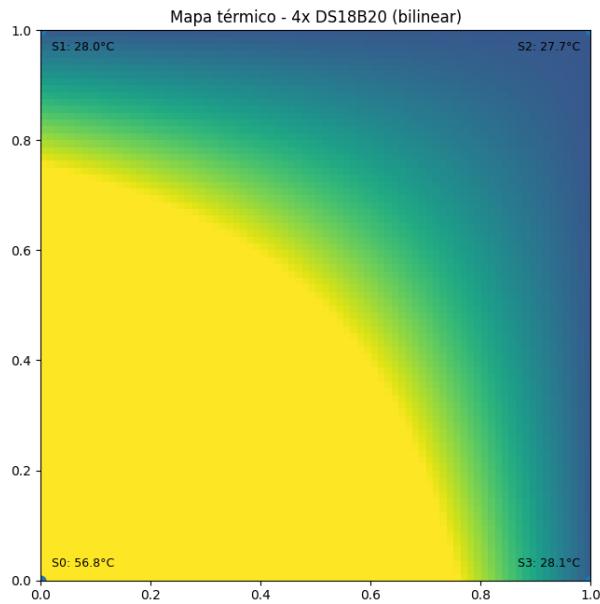


Figura 17: T1: Aquecimento em S0

7.6 T2 (resfriamento)

Retiramos o calor e capturamos a volta gradual às cores frias.

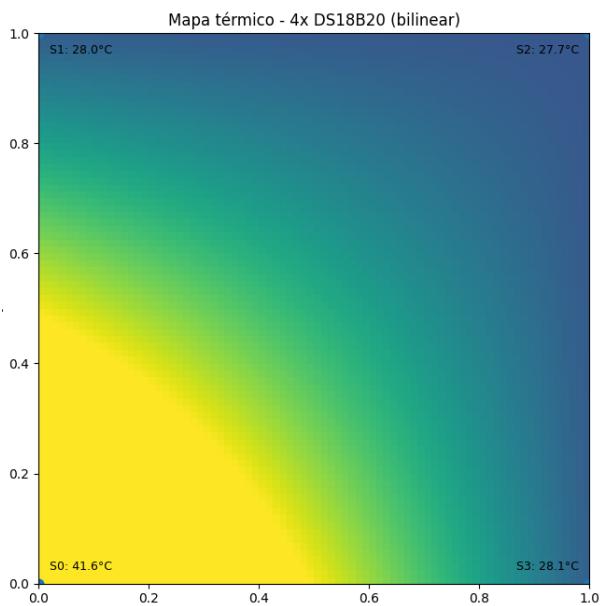


Figura 18: T2: Resfriamento

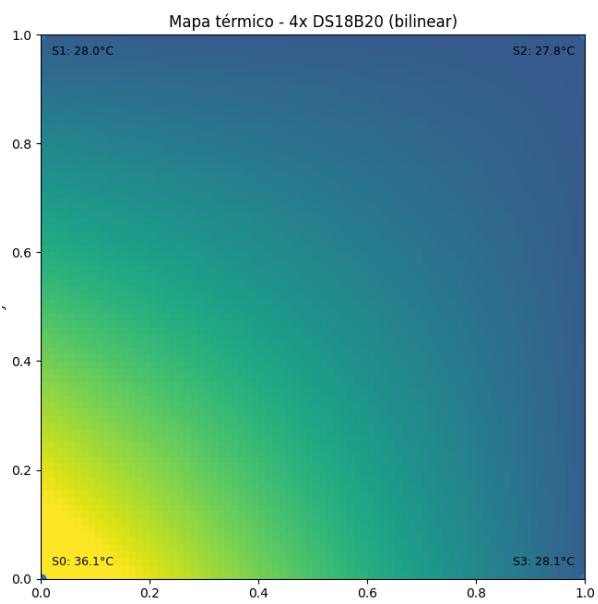


Figura 19: T2: Resfriamento

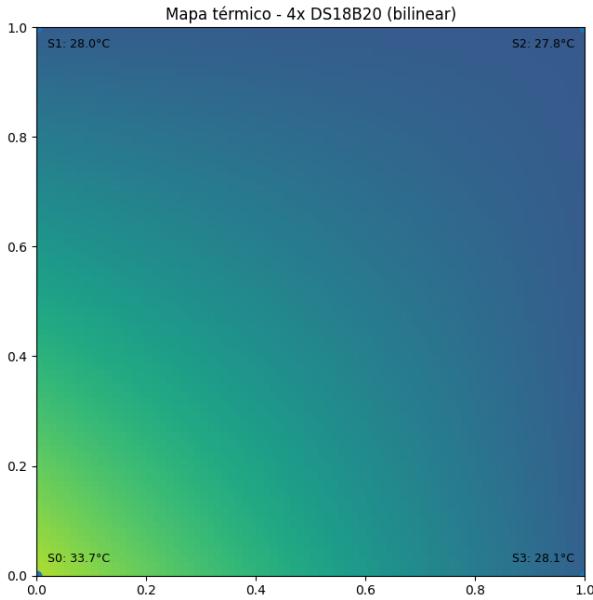


Figura 20: T2: Resfriamento

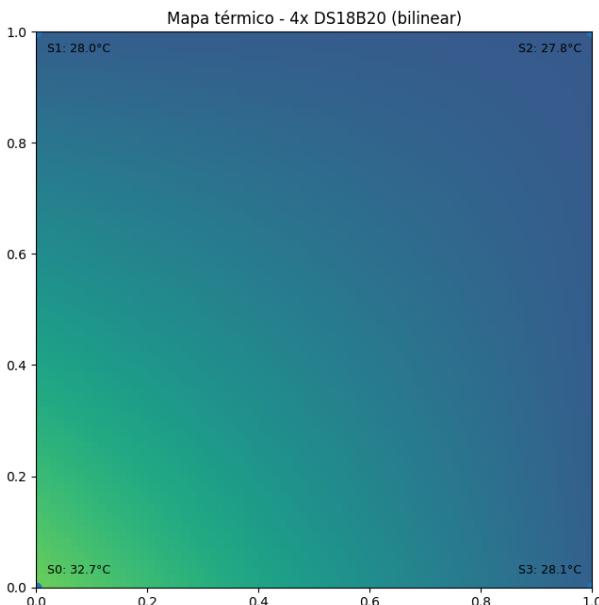


Figura 21: T2: Resfriamento

8 Resultados e discussões

8.1 Por que FPGA?

A escolha da FPGA se sustenta por seu paralelismo real, com um controlador 1-Wire por canal, os quatro sensores convertem em paralelo e são lidos de forma independente, sem disputar barramento. Para escalar para 8, 16 ou 32 canais significa instanciar mais controladores/bancos de I/O, preservando a mesma janela de 750 ms (12 bits) por ciclo — difícil de garantir, de forma determinística, em microcontroladores sob múltiplas tarefas.

8.2 Utilidade e aplicações reais

O arranjo é útil em cenários de pré-alarme térmico e tendência de aquecimento onde câmeras IR seriam exagero ou o layout impede linha de visada. Exemplos:

- **Painéis/quadros elétricos:** barramentos, disjuntores, terminais — antecipar pontos de aquecimento por mau contato.
- **Silos, estufas e câmaras frias:** gradientes de temperatura, fermentação localizada (silos) ou falha de refrigeração.
- **Data centers:** hot aisle/cold aisle em racks sem câmera térmica.
- **Monitoramento de incêndios:** O projeto poderia ser escalado para monitoramento de áreas nativas, como o cerrado, por exemplo, e assim atuar na prevenção de incêndios.

8.3 Limitações

O modelo apesar de ser útil em diversas aplicações reais, possui uma série de limitações, como sua resolução espacial, uma vez que quatro pontos não capturam curvaturas internas; mitigar com mais sensores ou modelos térmicos do processo. A própria rede 1-wire, cabos longos pedem pares trançados, stubs mínimos e, em parasite power, strong pull-up; aqui usamos alimentação normal a 3,3 V.

8.4 Expansões Futuras

Futuramente esse sistema pode ser expandido e aperfeiçoado por exemplo utilizando a própria FPGA para realizar o processamento de imagens e desenho do heatmap. Outra grande expansão seria utilizar modelos de Machine Learning para análise de históricos, anomalias e manutenção preditiva.

9 Anexos

9.1 Fotos do Sistema

9.1.1 Estrutura de Suporte (Impressora 3D)

Nessa estrutura colocamos um sensor ds18b20 em cada vértice do retângulo interior.

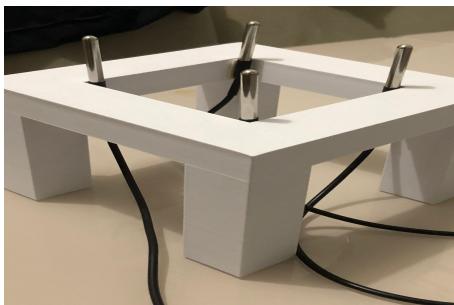


Figura 22: Estrutura de Suporte (Impressora 3D)

9.1.2 Display de 7 segmentos de 3 dígitos

Nessa protoboard conseguimos identificar os displays de 7 segmentos de três dígitos, com seus respectivos resistores e jumpers para ligações na pinagem da FPGA. Eles mostram a temperatura de cada sensor, na sequência S0, S1, S2 e S3.

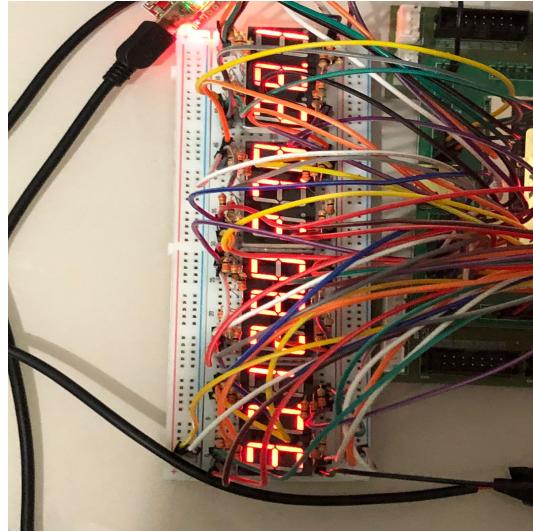


Figura 23: Display de 7 segmentos de 3 dígitos

9.1.3 Protoboard com Botão Reset e Ligação sensores ds18b20

Nessa protoboard conseguimos identificar as ligações dos sensores ds18b20 bem como - no meio da imagem - o botão de reset ativo em baixo.

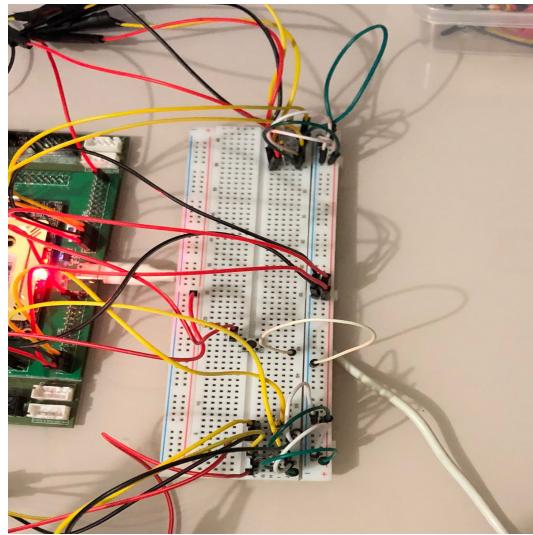


Figura 24: Botão reset e jumpers dos ds18b20

9.1.4 Ftdi ft232rl

Conversor Serial UART

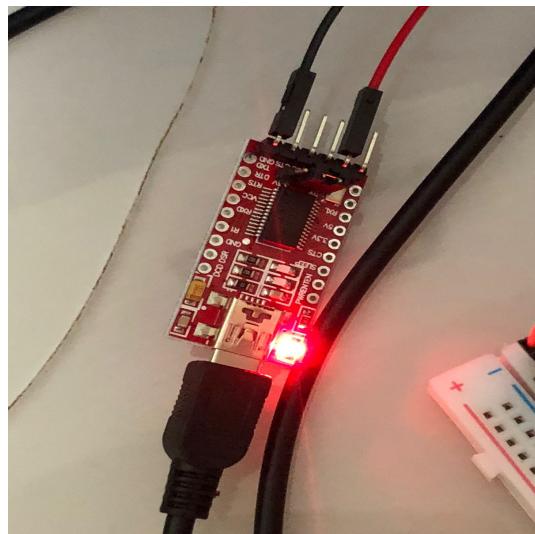


Figura 25: Ftdi ft232rl

9.1.5 FPGA Colorlight i9

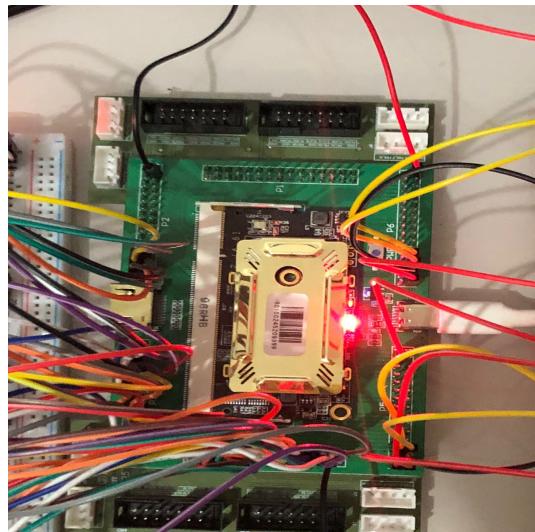


Figura 26: FPGA

9.1.6 FPGA + Protoboards

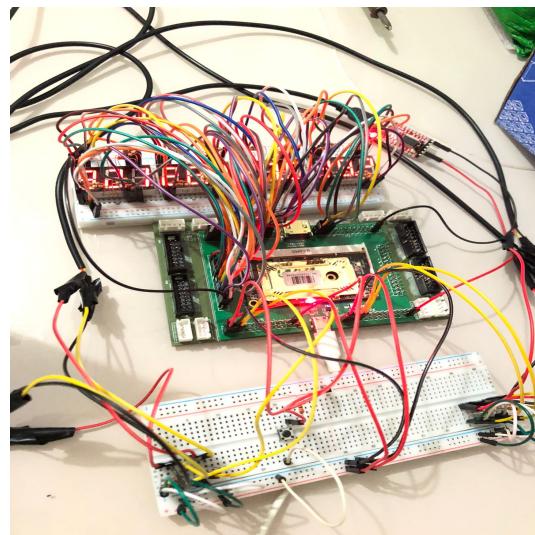


Figura 27: FPGA

9.1.7 Sistema Completo

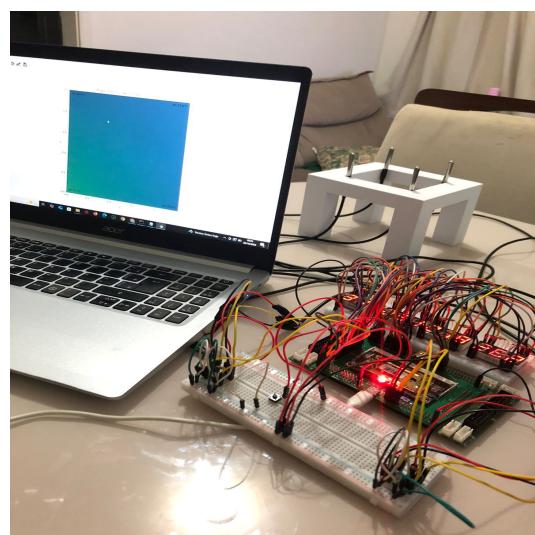


Figura 28: FPGA

9.2 Pinagem FPGA colorlight i9

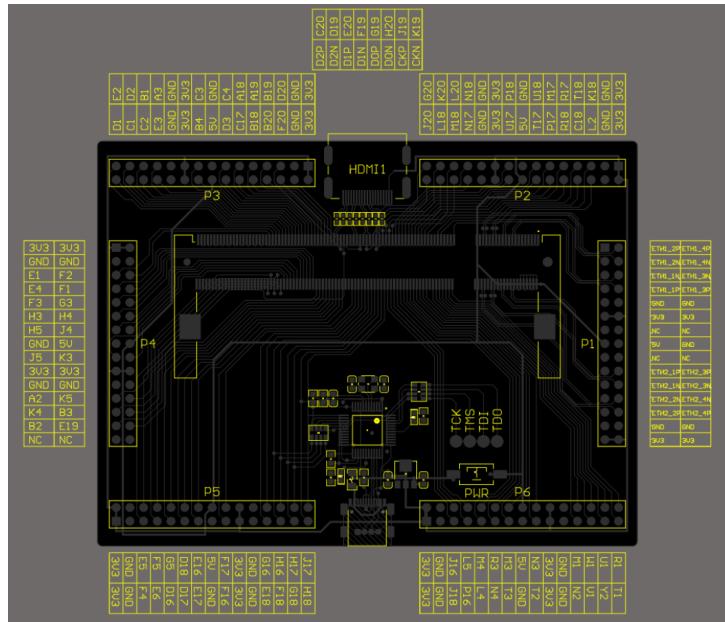


Figura 29: FPGA Colorlight i9

9.2.1 Display 7 segmentos de 3 dígitos (5631AS) Catodo Comum

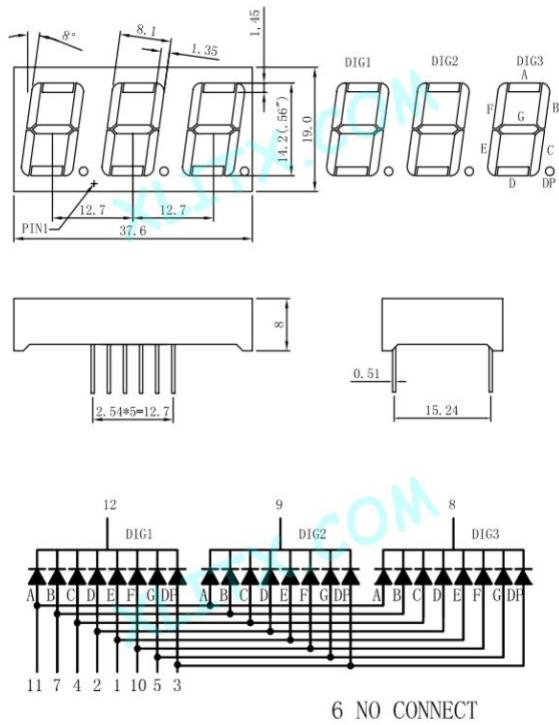


Figura 30: FPGA Colorlight i9

Tabela 1: Mapeamento dos sensores e seus respectivos pinos

SEGMENTO	NÚMERO PINO	DISPLAY 0	DISPLAY 1	DISPLAY 2	DISPLAY 3
A	11	J4	C1	A18	M18
B	7	K5	E1	C3	J20
C	4	B3	E4	A3	F20
D	2	E19	F3	B1	B20
E	1	B2	H4	C2	A19
F	10	H5	E2	C17	K20
G	5	A2	F1	B4	D20
PONTO	3	K4	G3	E3	B19
DIG 1	12	H3	D2	B18	L20
DIG 2	9	K3	D1	C4	L18
DIG 3	8	J5	F2	D3	G20

9.2.2 Sensores DS18B20

Tabela 2: Mapeamento dos sensores e seus respectivos pinos

SENSOR	PINO
SENSOR 0	F4
SENSOR 1	E5
SENSOR 2	J18
SENSOR 3	P16

9.2.3 Botão Reset

Alocamos o botão *reset_n* no pino H17, sua lógica é Ativo em baixo. Esse é fundamental para que na inicialização do sistema seja possível forçar o sistema para um estado conhecido com relação a seus elementos de memória (Flip-flops, registradores, etc).

9.3 GitHub do projeto

Todos os códigos, documentação, vídeos de apresentação do projeto e detalhes do desenvolvimento do projeto estão livremente disponíveis no GitHub, no repositório monitoramento_termico_FPGA:

<https://github.com/Zazamartins>

Referências

- [1] UART: Um protocolo de comunicação de hardware que comprehende o receptor/transmissor assíncrono universal. Disponível em: <https://www.analog.com/en/resources/analog-digital/articles/uart-a-hardware-communication-protocol.html>.
- [2] OneWire. Disponível em: <https://github.com/PaulStoffregen/OneWire>.
- [3] 1-wire (onewire) Master user: Disponível em: https://opencores.org/websvn/filedetails?repname=sockit_owm&path=%2Fsockit_owm%2Ftrunk%2Fdoc%2Fsockit_owr.pdf
- [4] UART (Universal Asynchronous Receiver Transmitter). Disponível em https://www.futek.com/uart-and-spi-in-embedded-systems?gad_source=1&gad_campaignid
- [5] UART, porta serial, interface RS-232. Disponível em <https://nandland.com/uart-serial-port-module/>
- [6] Interpolação Bilinear. Disponível em: <https://www.sciencedirect.com/topics/computer-science/bilinear-interpolation>
- [7] How to Perform Bilinear Interpolation in Python?. Disponível em: <https://www.askpython.com/python-modules/numpy/bilinear-interpolation-python>
- [8] alldatasheet. Disponível em: https://www.alldatasheet.com/view.jsp?Searchword=Ds18b20&gad_source=1&gad_campaignid=17
- [9] Programmable Resolution 1-Wire Digital Thermometer. Disponível em: <https://www.analog.com/media/en/technical-documentation/data-sheets/ds18b20.pdf>
- [10] FPGAEkey. Disponível em: <https://www.fpgakey.com/downloadfile/details/68?srsltid=AfmB0oq0ilzWh6BzbGp82cPKWpHBhRXAVzPFDjNQFBf2Crqq8v0f-Jin>