

IoTシステム科での1年間の活動まとめ

～ デジタル時計製作プロジェクトを中心に ～

東京都立中央・城北職業能力開発センター板橋校
IoTシステム科

2025年度

笠井 春希

目次

1. プロジェクト概要
2. 開発環境・使用技術
3. プロジェクト構成と開発フロー
4. システムアーキテクチャ（階層構造・ブロック図）
5. 時計製作：5つの機能
6. 主要モジュールの実装（ソースコード掲載）
7. テストベンチによる全網羅検証
8. C言語との連携
9. 技術的工夫8つのポイント
10. 的当てゲーム製作
11. まとめと今後の展望

1. プロジェクト概要

【目標】

2000年～2100年まで動作する101年対応デジタル時計の完全自力実装
5つの機能：時刻表示、時刻設定、アラーム、ストップウォッチ、タイマー
FPGAボード（Zybo Z7-20）と4桁7セグメントLEDによる実装

【開発期間・規模】

開発期間：約2ヶ月（2024年9月～11月）
コード規模：16モジュール、約850行（Verilog HDL）
検証規模：123,290パターンの全網羅テスト（86,400 + 36,890）

【開発方針】

段階的开发：24時間 → 年月日曜日 → 統合、各フェーズで完全にテスト

2. 開発環境・使用技術

【ハードウェア】

時計製作：

FPGAボード：Zybo Z7-20
(Xilinx Zynq-7000)

LEDディスプレイ：

4桁7セグメントLED

入力：4個のプッシュボタン

開発環境：Vivado 2023.2

HDL：Verilog HDL

検証補助：C言語 (gcc)

シミュレータ：

Vivado Simulator

的当てゲーム：

FPGAボード：Cmod A7
(Xilinx Artix-7)

LEDマトリクス：32×32 RGB LED

ラズベリーパイ4 + カメラ

【ソフトウェア】

開発環境：Vivado 2023.2

HDL：Verilog HDL

検証補助：C言語 (gcc)

シミュレータ：

Vivado Simulator

【使用技術】

階層設計・モジュール分割、carry信号連鎖によるカウンタ設計

ブロックRAMを用いた曜日計算の最適化 (Critical Path短縮)

ダイナミック点灯制御 (デコーダ4個→1個、データ幅32bit→16bit)

メタステビリティ対策・チャタリング対策・エッジ検出の統合実装

C言語による大規模リファレンスデータ生成とテストベンチ連携

境界値テスト6パターンによる厳密な検証

3. プロジェクト構成と開発フロー

【3段階の開発プロセス】

Phase 1: 24時間プロジェクト（時・分・秒）

CNT60.v（秒・分カウンタ、0-59カウント、carry出力）

CNT24.v（時カウンタ、0-23カウント）

CountReference.c で86,400パターンのリファレンス生成 → 全網羅テスト成功

Phase 2: 年月日曜日プロジェクト（年・月・日・曜日）

CNT_YEAR.v（年カウンタ、2000-2100年、101年ループ）

CNT_MONTH.v（月カウンタ、1-12月）、CNT_DAY.v（日カウンタ、月ごとの日数対応）

weekday_calc.v（曜日計算、ブロックRAM使用）

DMY_ref.c で36,890パターンのリファレンス生成 → 全網羅テスト成功

Phase 3: 統合プロジェクト（全機能統合）

CLOCK_ALL.v（トップモジュール）

MAIN_MODE.v（動作モード管理）、SET_MODE.v（設定処理）、DIS_MODE.v（表示切替）

DECODER7.v（7セグデコーダ）、DCOUNT.v（ダイナミック点灯制御）

MSE.v（メタステビリティ + チャタリング + エッジ検出）

Phase1とPhase2の成果を統合し、5機能を実装

4-1. システムアーキテクチャ（階層構造）

【モジュール階層】

```
CLOCK_ALL (トップモジュール)
├─ SEC1 (1秒クロック生成)
├─ MAIN_MODE (動作モード管理：時刻表示/設定/アラーム/SW/タイマ
一)
├─ SET_MODE (設定モード処理)
├─ DIS_MODE (表示切替処理)
├─ カウンタ群：
│   ├── CNT_YEAR (年カウンタ、2000-2100年、101年ループ)
│   ├── CNT_MONTH (月カウンタ、1-12月、うるう年対応)
│   ├── CNT_DAY (日カウンタ、1-31日、月ごとの日数対応)
│   ├── weekday_calc (曜日計算、ブロックRAM 1212エントリ)
│   ├── leap_year (うるう年判定)
│   ├── CNT24 (時カウンタ、0-23時)
│   └── CNT60 x2 (分・秒カウンタ、0-59)
├─ 表示制御群：
│   ├── display_switch (表示データ切替)
│   ├── DECODER7 (7セグメントデコーダ)
│   └── DCOUNT (ダイナミック点灯制御、4桁LED切替)
├─ 入力処理：
│   └── MSE x4 (メタステビリティ + チャタリング + エッジ検出)
```

合計：16モジュール、約850行

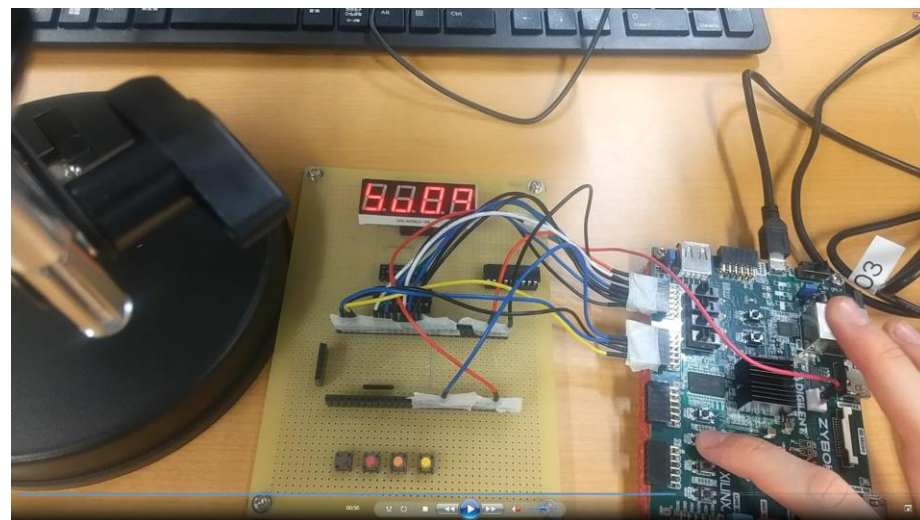
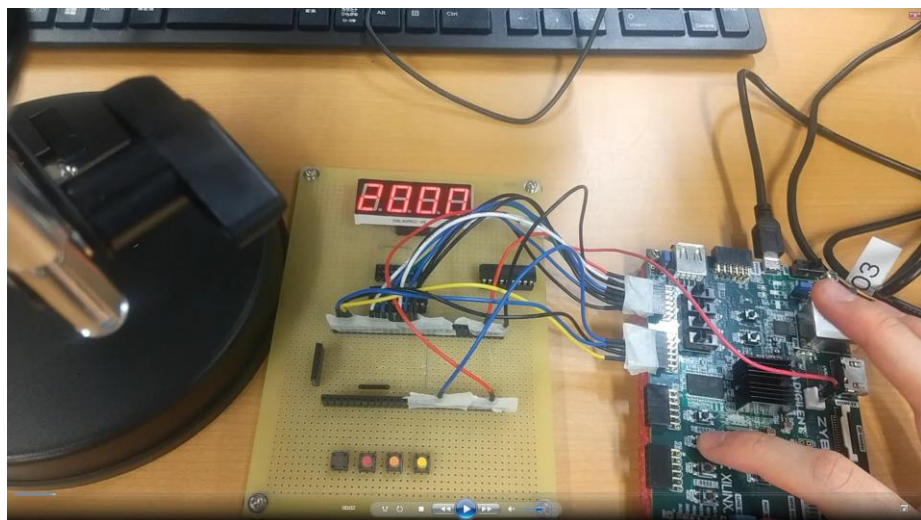
4-2. ハードウェアブロック図

| | |
|---|--|
| トップモジュール CLOCK_ALL | |
| [入力] 4個のボタン → MSE x4 (安定化) [クロック] 125MHz → SEC1 → ENABLE(1秒ごとに1clkだけHigh) | |
| [モード制御] MAIN_MODE: 時刻表示/設定/アラーム/SW/タイマー SET_MODE: 設定中の桁を個別変更 DIS_MODE: 表示データ切替 | |
| [カウンタチェーン (carry連鎖)] CNT60(秒) -carry→ CNT60(分) -carry→ CNT24(時) -carry→ CNT_DAY(日) -carry→ CNT_MONTH(月) -carry→ CNT_YEAR(年) weekday_calc(曜日): ブロックRAM参照 leap_year(うるう年): 組み合わせ回路 | |
| [表示制御] display_switch → DECODER7 → LED[7:0] DCOUNT → SA[3:0] (桁選択信号) ダイナミック点灯: 1桁ずつ高速切替 | |



4桁7セグメントLED表示

4-3.【実機写真】



時刻表示例：20:00
Zybo Z7-20+ 4桁7セグLED

曜日表示：SU（日曜日）
7セグオリジナルパターン

5. 時計製作：5つの機能

① 時刻表示モード

- ・2000/01/01 00:00:00 (土) から 2100/12/31 23:59:59 (木) まで対応
- ・年月日・曜日・時分秒をリアルタイム表示、うるう年自動判定
- ・101年間のループ動作（2100年末 → 2000年初にロールオーバー）

② 時刻設定モード

- ・年・月・日・時・分・秒を個別に設定可能
- ・設定中の桁を点滅表示（DIS_MODEで制御）
- ・ボタンでインクリメント/デクリメント、設定完了で通常動作に戻る

③ アラームモード（実装予定）

- ・指定時刻にアラーム音を鳴らす機能

④ ストップウォッチモード（実装予定）

- ・0.01秒単位の計測、スタート/ストップ/リセット機能

⑤ タイマーモード（実装予定）

- ・カウントダウнтаイマー、時間設定・開始・停止機能

6-1. 主要モジュール① : CNT60.v (60進カウンタ)

【ENABLEによる同期制御のタイミング】

CLK: システムクロック(125MHz)

ENABLE: 1秒ごとに1clkだけHigh (tmp_count==125M-1の瞬間)

【3つのCARRY信号の役割】

CARRY_in: 上位から受け取る更新許可信号

→ 秒カウンタなら常に1、分カウンタなら秒のCARRY_out

CARRY: 内部信号(組み合わせ回路で生成)

→ $CARRY = (CNT10 == 9) \ \&\& \ CARRY_in$

CARRY_out: 下位に送る更新許可信号

→ $CARRY_out = (CNT6 == 5) \ \&\& \ CARRY$

→ 次段のカウンタのCARRY_inになる

【更新タイミング】

CLKの立ち上がりで (ENABLE && CARRY_in) を判定

→ 全カウンタが同じCLKエッジで同時に判定・更新

【60進カウンタ:0-59をカウント、59でcarry信号出力】

```
module CNT60(RESET, CLK, CNT6, CNT10, ENABLE, CARRY_in, CARRY_out, SET_CURRENT_STATE, INC_MODE);
input RESET, CLK, ENABLE, CARRY_in, INC_MODE;
input [1:0] SET_CURRENT_STATE;
output reg CARRY_out;
output [3:0] CNT10;
output [3:0] CNT6;
reg [3:0] CNT10;
reg [3:0] CNT6;
reg CARRY;

always @(posedge CLK or posedge RESET)
begin
    if (RESET == 1'b1)
        begin
            CNT10 <= 4'h0;
        end
    else if (((ENABLE == 1'b1 && CARRY_in == 1'b1) && SET_CURRENT_STATE[0] == 1'b1) ||
            (SET_CURRENT_STATE[1] == 1'b1 && INC_MODE == 1'b1))
        begin
            if (CARRY == 1'b1)
                CNT10 <= 4'h0;
            else
                CNT10 <= CNT10 + 4'h1;
        end
    end

always @(CNT10 or CARRY_in or SET_CURRENT_STATE or INC_MODE)
begin
    if (CNT10 == 4'h9 && (CARRY_in == 1'b1 || (SET_CURRENT_STATE[1] == 1'b1 && INC_MODE == 1'b1)))
        CARRY <= 1'b1;
    else
        CARRY <= 1'b0;
    end
end
```

【60進カウンタ:0-59をカウント、59でcarry信号出力】

```
end

always @(CNT6 or CARRY)
begin
    begin
        if (CNT6 == 4'h5 && CARRY == 1'b1)
            CARRY_out <= 1'b1;
        else
            CARRY_out <= 1'b0;
        end
    end
end

always @(posedge CLK or posedge RESET)
begin
    if (RESET == 1'b1)
        begin
            CNT6 <= 3'b000;
        end
    else if ((CARRY == 1'b1) && ((ENABLE == 1'b1 && SET_CURRENT_STATE[0] == 1'b1) ||
        (SET_CURRENT_STATE[1] == 1'b1 && INC_MODE == 1'b1)))
        begin
            if (CNT6 == 3'b101)
                CNT6 <= 3'b000;
            else
                CNT6 <= CNT6 + 3'b001;
            end
        end
    end
end

endmodule
```

6-2. 主要モジュール② : CNT24.v (24時間カウンタ)

【ENABLEによる同期制御のタイミング】

CLK: システムクロック(125MHz)

ENABLE: 1秒ごとに1clkだけHigh (tmp_count==125M-1の瞬間)

【3つのCARRY信号の役割】

CARRY_in: 上位から受け取る更新許可信号

→ 時カウンタなら分のCARRY_out

CARRY: 内部信号(組み合わせ回路で生成)

→ $CARRY = (CNT10 == 9 \mid \mid \{CNT3, CNT10\} == 23) \&\& CARRY_in$

→ 23時の場合は特別処理

CARRY_out: 下位に送る更新許可信号

→ $CARRY_out = (CNT3 == 2) \&\& CARRY$

→ 次段(日カウンタ)のCARRY_inになる

【更新タイミング】

CLKの立ち上がりで (ENABLE && CARRY_in) を判定

→ {CNT3, CNT10}=23の次は00に戻る

【24時間カウンタ：0-23をカウント、23でcarry信号出力】

```
module CNT24(RESET, CLK, CNT3, CNT10, ENABLE, CARRY_in, CARRY_out, SET_CURRENT_STATE, INC_MODE);
input RESET, CLK, ENABLE, CARRY_in, INC_MODE;
input [1:0] SET_CURRENT_STATE;
output reg CARRY_out;
output [3:0] CNT10;
output [3:0] CNT3;
reg [3:0] CNT10;
reg [3:0] CNT3;
reg CARRY;
wire [3:0] max10;

always @(posedge CLK or posedge RESET)
begin
    if (RESET == 1'b1)
        begin
            CNT10 <= 4'h0;
        end
    else if (((ENABLE == 1'b1 && CARRY_in == 1'b1) && SET_CURRENT_STATE[0] == 1'b1) ||
            (SET_CURRENT_STATE[1] == 1'b1 && INC_MODE == 1'b1)) begin

        begin
            if (CARRY == 1'b1)
                CNT10 <= 4'h0;
            else
                CNT10 <= CNT10 + 4'h1;
        end
    end
end

always @(CNT10 or CARRY_in or SET_CURRENT_STATE or INC_MODE)
begin
    begin
        if ((CNT10 == 4'h9 || {CNT3,CNT10} == 8'h23) &&
            (CARRY_in == 1'b1 || (SET_CURRENT_STATE[1] == 1'b1 && INC_MODE == 1'b1)))
            CARRY <= 1'b1;
    end
end
```

【24時間カウンタ：0-23をカウント、23でcarry信号出力】

```
        else
            CARRY <= 1'b0;
        end
    end

    always @(CNT3 or CARRY)
    begin
        begin
            if (CNT3 == 4'h2 && CARRY == 1'b1)
                CARRY_out <= 1'b1;
            else
                CARRY_out <= 1'b0;
            end
        end
    end

    always @(posedge CLK or posedge RESET)
    begin
        if (RESET == 1'b1)
            begin
                CNT3 <= 2'b00;
            end
        else if ((CARRY == 1'b1) && ((ENABLE == 1'b1 && SET_CURRENT_STATE[0] == 1'b1) ||
            (SET_CURRENT_STATE[1] == 1'b1 && INC_MODE == 1'b1)))
            begin
                begin
                    if (CNT3 == 2'h2)
                        CNT3 <= 2'h0;
                    else
                        CNT3 <= CNT3 + 2'h1;
                    end
                end
            end
        end
    end

endmodule
```

6-3. 主要モジュール③ : weekday_calc.v (曜日計算)

【weekday_calc.v - 曜日計算モジュール】

課題 : 年月日から曜日 (0=日曜~6=土曜) を計算

従来の方法の問題点 :

- Zellerの公式など複雑な計算式
- 除算・剰余演算が必要
- FPGAで実装すると大量のLUTを消費
- 計算に複数サイクル必要

採用した解決策 : Block RAM (BRAM) によるルックアップテーブル

- 年×12ヶ月 (2000-2100年) のデータを事前計算してBRAMに格納
- アドレス計算: $\text{year} \times 12 + (\text{month} - 1)$ で1日の曜日を取得
- 実際の日の曜日: $(\text{bram_data} + \text{day} - 1) \% 7$

実装の特徴 :

✓ 掛け算を使わずにアドレス計算

$\text{year} \times 12 = (\text{year} \ll 3) + (\text{year} \ll 2) \ // \ 8 * \text{year} + 4 * \text{year}$

✓ 1サイクルでBRAMアクセス完了

✓ LUT使用量を大幅削減

```
module weekday_calc(
    input [11:0] year_bcd, // 3桁BCD
    input [7:0] month_bcd, // 2桁BCD
    input [7:0] day_bcd, // 2桁BCD
    input clk,
    output reg [3:0] weekday // 0=日曜~6=土曜
);
    wire [11:0] bram_addr = (year_bin << 3) + (year_bin
    << 2) + (month_bin - 1);
    always @(posedge clk)
        weekday <= (bram_dout + day_bin - 1) % 7;
endmodule
```


【ブロックRAM使用：各月1日の曜日をルックアップ、日数を加算】

```
module weekday_calc(
    input  wire [11:0] year_bcd,
    input  wire [7:0]  month_bcd,
    input  wire [7:0]  day_bcd,
    input  wire        clk,
    output reg [3:0]   weekday
);

    wire [11:0] year_bin;
    wire [7:0]  month_bin;
    wire [7:0]  day_bin;

    assign year_bin = ((year_bcd[11:8] << 6) + (year_bcd[11:8] << 5) + (year_bcd[11:8] << 2) +
                      (year_bcd[7:4]  << 3) + (year_bcd[7:4]  << 1) +
                      year_bcd[3:0]);

    assign month_bin = ((month_bcd[7:4] << 3) + (month_bcd[7:4] << 1) + month_bcd[3:0]);
    assign day_bin   = ((day_bcd[7:4]   << 3) + (day_bcd[7:4]   << 1) + day_bcd[3:0]);

    wire [11:0] bram_addr;
    assign bram_addr = (year_bin << 3) + (year_bin << 2) + (month_bin - 1);

    wire [3:0] bram_dout;
    blk_mem_gen_0 weekday_bram (
        .clka(clk),
        .ena(1'b1),
        .wea(1'b0),
        .addra(bram_addr),
        .douta(bram_dout)
    );

    always @(posedge clk) begin
        weekday <= (bram_dout + day_bin - 1) % 7;
    end

endmodule
```

6-4. 主要モジュール④ : MSE.v (統合入力処理)

【MSE.v - 統合入力処理モジュール】

MSE = Metastability + Switch(Chattering) + Edge の統合処理

【3つの問題を同時に解決】

1. Metastability (メタステーブル) 対策
 - 非同期入力信号をクロックドメインに取り込む際の不安定状態
 - 解決策: 2段フリップフロップ (META[1:0]) でサンプリング
2. Switch Chattering (チャタリング) 除去
 - 機械式スイッチの接点が安定するまでの短時間のON/OFF繰り返し
 - 解決策: 4ビットシフトレジスタ (SFT[3:0]) で連続4回一致を確認
 - ENABLE_kHzで約1ms間隔でサンプリング
3. Edge Detection (エッジ検出)
 - ボタンの「押した瞬間」だけ1になる信号を生成
 - 解決策: 2段フリップフロップ (ED[1:0]) で立ち上がりエッジ検出
 - MODE = ED[0] & ~ED[1]

【モジュールの実装】

```
module MSE(CLK, MODE_IN, MODE, ENABLE_kHz);
    input CLK, MODE_IN, ENABLE_kHz;
    output MODE;
    reg [1:0] META; // メタステーブル対策
    reg [3:0] SFT; // チャタリング除去
    reg [1:0] ED; // エッジ検出

    always @(posedge CLK) META <= {META[0], MODE_IN};
    always @(posedge CLK) if(ENABLE_kHz) SFT <= {SFT[2:0],
META[1]};
    wire CHATA = & SFT; // 4ビット全て1
    always @(posedge CLK) ED <= {ED[0], CHATA};
    assign MODE = ED[0] & ~ED[1];
endmodule
```

【設計の利点】

- ✓ 3つの機能を1つのモジュールに統合
- ✓ すべてのボタン入力に共通して使用可能
- ✓ 信頼性の高い入力処理を実現

7-1. テストベンチによる全網羅検証①：設計思想

【検証方針】

すべての時刻パターンを1つずつテストする「全網羅検証」

- 24時間プロジェクト：86,400パターン（00:00:00 ～ 23:59:59）
- 年月日曜日プロジェクト：36,890パターン（2000/01/01 ～ 2100/12/31）

→ 人間が手作業で確認することは不可能
→ 自動化された検証手法が必須

【テストベンチの仕組み】

- ① C言語で正解データ（リファレンスファイル）を事前生成
→ 人間が手で確認できない規模を自動化
→ BCD形式のカウント値を全パターン列挙
- ② Verilogテストベンチでリファレンスファイルを読み込み
→ \$readmemh()でメモリに読み込み
→ 1サイクルごとにVerilog出力値と比較
- ③ 不一致があればエラー表示して停止
→ どのステップで失敗したかを明確に報告
→ デバッグが容易
- ④ 全パターン一致したら成功メッセージ
→ "Total OK steps = 86400" など

【検証の重要性】

- ✓ 境界値（23:59:59→00:00:00など）での動作確認
- ✓ 閏年処理の正確性確認
- ✓ 曜日計算の正確性確認（101年分）
- ✓ すべてのパターンで正常動作を保証

7-2. テストベンチによる全網羅検証②：タイミング制御

【タイミング制御の核心部分】

```
// 24時間テストベンチ (TEST_CNT246060_ALL.v) の検証ループ
for (i = 1; i < MAX_NUM; i = i + 1) begin
    @(negedge i1.ENABLE);          // ① ENABLEの立ち下がりを待つ
    cnt_value_ref = ref[i];        // ② リファレンスデータを先に読み込む
                                    // ※この時点では回路はまだ動いていない

    @(posedge i1.ENABLE);          // ③ ENABLEの立ち上がりを待つ
                                    // ※ここで回路が動作し、カウンタが更新される

    @(negedge clk);                // ④ clkの立ち下がりデータ確定を待つ
                                    // ※出力が安定してから比較

    if (cnt_value != cnt_value_ref) begin // ⑤ 比較・検証
        $display("Error at step %d: cnt_value=%X expected=%X",
            i, cnt_value, ref[i]);
        $display("Total OK steps = %d", ok_count);
        $stop;
    end else
        ok_count = ok_count + 1;
end

// この順序により、リファレンスが先→実際の動作→検証の順が保証される
```

【ENABLEによる同期制御のタイミング】

テストベンチでの検証タイミング:

【初回のみ(ループ外)】

1. @(posedge ENABLE) : 最初のENABLE立ち上がりを待つ
2. @(negedge clk) : clkの立ち下がりで値検証

【ループ内(2回目以降)】

1. @(negedge ENABLE) : ENABLEの立ち下がり検出
2. リファレンスデータを先読み(cnt_value_ref = ref[i])
3. @(posedge ENABLE) : ENABLEの立ち上がり(カウンタ更新)
4. @(negedge clk) : clkの立ち下がりで値検証

【初回をループ外で処理する理由】

- ・clkの初期状態はLow
- ・ループ内で @(negedge clk) を使うと、初回は即座に検出されてしまう
- ・初回だけループ外で処理することで、正確なタイミング制御を実現

7-3. テストベンチによる全網羅検証③：ソースコード

【TEST_CNT246060_ALL.v（24時間プロジェクト用テストベンチ）】

```
module TEST_CNT246060_ALL;
    parameter MAX_NUM = 60*60*24; // 86,400パターン
    reg clk, reset, dec, btn_switch;
    wire [7:0] led;
    wire [3:0] sa;
    reg [23:0] ref [0:MAX_NUM - 1]; // リファレンスデータ配列
    reg [23:0] cnt_value, cnt_value_ref;
    integer i;
    reg [16:0] ok_count;

    parameter CYCLE = 100;
    parameter SIM_SEC1_MAX = 4; // シミュレーション高速化

    // DUT (Device Under Test) インスタンス
    CNT246060_ALL #(.SEC1_MAX(SIM_SEC1_MAX)) i1(
        .RESET(reset), .CLK(clk), .DEC(dec),
        .btn_switch(btn_switch), .LED(led), .SA(sa)
    );

    // カウンタ値を読み出し
    always @(posedge clk) begin
        cnt_value = {2'b0, i1.CNT3, i1.CNT10_3,
                    1'b0, i1.CNT6_2, i1.CNT10_2,
                    1'b0, i1.CNT6, i1.CNT10};
    end

    always #(CYCLE/2) clk = ~clk; // クロック生成

    initial begin
        $readmemh("ref.hex", ref); // C言語生成のリファレンス読込
    end

    // 検証ループは前スライド参照
endmodule
```

7-4. テストベンチによる全網羅検証④：===演算子の活用

【不定値（X）を含む厳密な比較による正確な検証】

【問題点】

- 最初は == 演算子で比較
- X（不定値）と 0 の比較で true が返される
- テストベンチが正しいエラーを検出できない

【解決策：=== 演算子の使用】

- === 演算子（厳密等価比較）を使用
- X（不定値）、Z（ハイインピーダンス）も含めて厳密に比較
- 不定値が含まれる場合は false を返す

【成果】

- 正確な検証結果を得られるようになった
- 初期化忘れや信号の不定値を確実に検出可能
- 24時間（86,400パターン）および年月日（36,890パターン）の全網羅検証が成功

8-2. C言語との連携①：年月日曜日リファレンス生成

【CountReference.c：86,400パターンのリファレンス生成】

```
#include <stdio.h>

int main(void) {
    int h, m, s, hh, mm, ss;
    FILE* fp = fopen("ref.hex", "w");
    if (!fp) return 1;

    // 00:00:00 から 23:59:59 まで全パターン生成
    for (h = 0; h < 24; h++) {
        for (m = 0; m < 60; m++) {
            for (s = 0; s < 60; s++) {
                // BCD形式に変換
                hh = ((h / 10) << 4) | (h % 10);
                mm = ((m / 10) << 4) | (m % 10);
                ss = ((s / 10) << 4) | (s % 10);

                // 16進数で出力（例：23:59:59 → 235959）
                fprintf(fp, "%02X%02X%02X\n", hh, mm, ss);
            }
        }
    }
    fclose(fp);
    return 0;
}

// 実行結果：ref.hex ファイルに86,400行のデータ
// 000000, 000001, 000002, ..., 235958, 235959
```


8-2. C言語との連携②：年月日曜日リファレンス生成

【DMY_ref.c：36,890パターンのリファレンス生成】

```
#include <stdio.h>

int main(void) {
    int year, month, day;
    int yy, mm, dd, max_day;
    int week_day = 6; // 2000/1/1は土曜
    FILE* fp = fopen("ref.hex", "w");

    for (year = 0; year <= 100; year++) {
        for (month = 1; month <= 12; month++) {
            // 月ごとの最大日数
            max_day = (month == 4 ||
                       month == 6 ||
                       month == 9 ||
                       month == 11) ? 30 :
                (month == 2) ?
                    ((year % 4 == 0) ?
                     ((year % 100 == 0) ?
                      ((year == 0) ? 29 : 28)
                      : 29)
                     : 28)
                    : 31;

            for (day = 1; day <= max_day; day++) {
```

```
                // 曜日を含めて出力
                // フォーマット：曜日(4bit) + 年(12bit)
                //                      + 月(8bit) + 日(8bit)
                fprintf(fp, "%01X%03X%02X%02X\n",
                        week_day, yy, mm, dd);

                week_day = (week_day + 1) % 7;
            }
        }
    }
    fclose(fp);
    return 0;
}
```

8-3. C言語との連携③：ブロックRAM用データ生成

【weekday.c → weekday.txt → weekday_coe.c → weekday.coe】

weekday.c (ツエラーの公式で曜日計算)

```
// 2000年～2100年の各月1日の曜日を計算
// 合計 101年 × 12ヶ月 = 1212エントリ
```

```
int zeller(int y, int m, int d) {
    int i, j, goukei = 0, w;
```

```
// 1年1月1日からの経過日数を計算
```

```
for (i = 1; i < y; i++) {
    for (j = 1; j <= 12; j++) {
        goukei += daysofmonth(i, j);
    }
}
```

```
for (j = 1; j < m; j++) {
    goukei += daysofmonth(y, j);
}
```

```
w = (goukei + d) % 7;
return w; // 0=日, 1=月, ..., 6=土
}
```

```
// 実行結果：weekday.txt
// 6 (2000/1/1は土曜)
// 2 (2000/2/1は火曜)
// ...
```

weekday_coe.c (.coeファイル生成)

```
// weekday.txt を読み込んで
// Vivado用の.coeファイルに変換
```

```
int main(void) {
    int val, count = 0;
    FILE *fin = fopen("weekday.txt", "r");
    FILE *fout = fopen("weekday.coe", "w");
```

```
fprintf(fout,
    "memory_initialization_radix=16;¥n");
fprintf(fout,
    "memory_initialization_vector=¥n");
```

```
while (fscanf_s(fin, "%d", &val) == 1) {
    fprintf(fout, "%d", val);
    count++;
    if (count < 1212) fprintf(fout, ",");
    if (count % 16 == 0)
        fprintf(fout, "¥n");
}
```

```
fprintf(fout, "¥n");
fclose(fin);
fclose(fout);
return 0;
}
```

9-1. 技術的工夫①: 条件信号による同期カウンタ設計

【設計思想】

各カウンタが「いつ更新すべきか」を組み合わせ回路で判定 上位カウンタは下位のCARRY信号を条件として、同じクロックエッジで判定

- 論理的な依存関係はあるが、時間的な遅延はゼロ
- 全カウンタが同じタイミングで動作、デバッグが容易

【3つのCARRY信号の関係】

秒カウンタ(CNT60):

```
CARRY_in = 1'b1(常に更新可能)
CARRY = (CNT10==9) && CARRY_in // 組み合わせ回路
CARRY_out = (CNT6==5) && CARRY // 組み合わせ回路
```

分カウンタ(CNT60):

```
CARRY_in = 秒のCARRY_out
CARRY = (CNT10==9) && CARRY_in
CARRY_out = (CNT6==5) && CARRY
```

時カウンタ(CNT24):

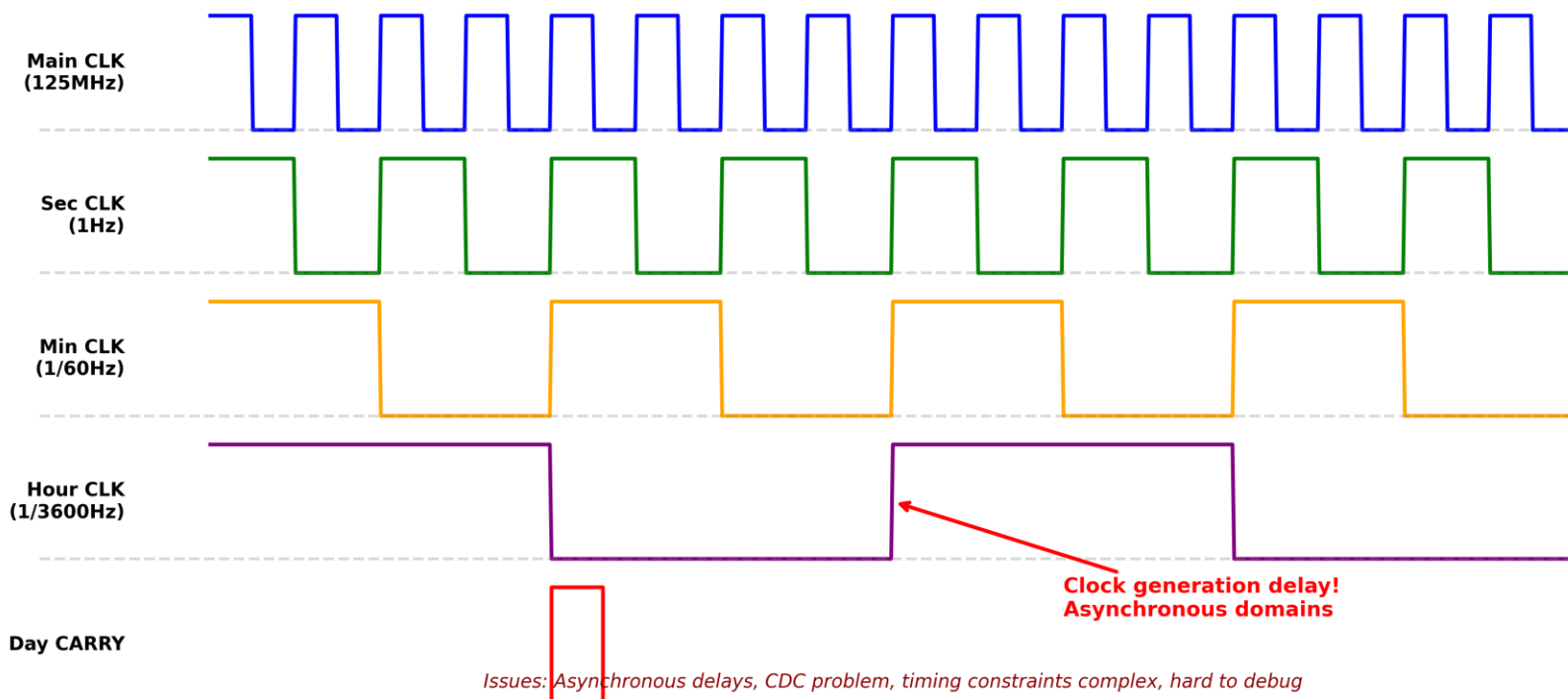
```
CARRY_in = 分のCARRY_out
CARRY = (CNT10==9 || {CNT3,CNT10}==23) &&
CARRY_in
CARRY_out = (CNT3==2) && CARRY
```

【重要ポイント】

すべてのCARRY信号は組み合わせ回路
(assign文またはalways @(*))
→ 同じCLKエッジで全カウンタが判定可能

9-1-1. 分周回路方式（×採用しなかった方式）

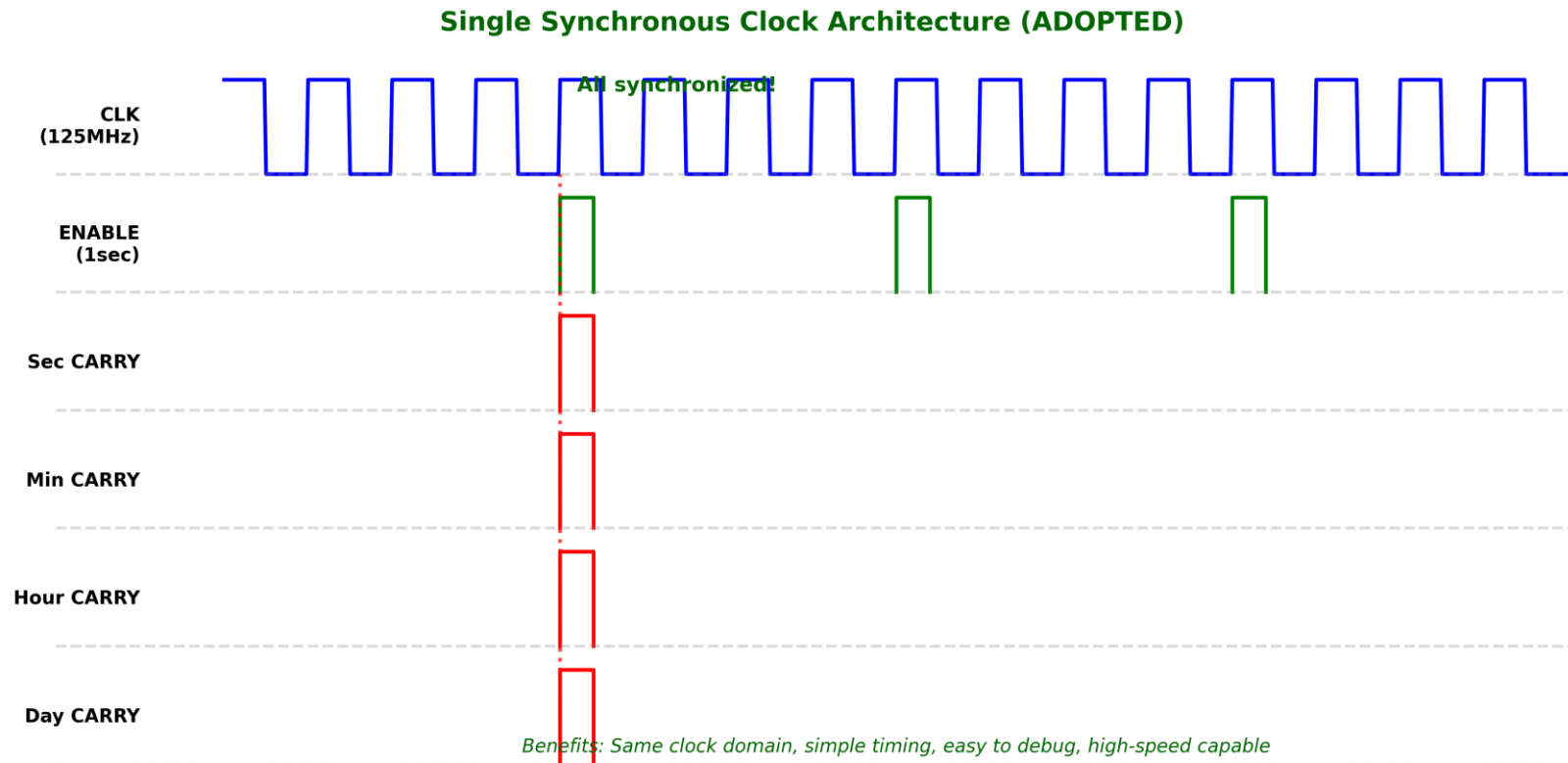
Distributed Clock Architecture (NOT ADOPTED)



【問題点】

- ・各段で異なるクロック周波数を使用（分周により生成）
 - 秒: 1Hz、分: 1/60Hz、時: 1/3600Hz
- ・異なるクロック間でのデータ転送時にタイミングずれが発生
 - 秒が「59→0」になった瞬間を分カウンタが受け取るのに最悪1サイクル(1秒)の遅延が生じる可能性
 - 正確な時刻が保証できない
- ・タイミング制約が複雑（クロック間の関係をすべて定義必要）
- ・デバッグが困難（どこでずれたか特定しにくい）
- ・高速動作に不向き（遅いクロックがボトルネックになる）

9-1-2. 単層同期回路方式 (☒ 採用した方式)



【利点】

- ・全モジュールが同じCLK(125MHz)で動作
- ・ENABLE信号をassign文で生成(組み合わせ回路)
- ・全てのCARRY信号も組み合わせ回路で生成
 - 秒→分→時の判定が同じCLKエッジで同時完了
 - 時間的な遅延ゼロ、正確な時刻を保証
- ・タイミング制約がシンプル(CLKの周期だけ指定すればOK)
- ・デバッグが容易(波形を見れば全信号が一目瞭然)
- ・高速動作が可能(125MHzで安定動作)

9-1-3. なぜ単層同期回路を採用したか

【設計方式の比較】

✕ 分周回路方式（不採用）

- 各段で異なるクロック周波数
- 秒CLK (1Hz) → 分CLK (1/60Hz) → 時CLK (1/3600Hz)
- クロック生成による遅延が発生
- 非同期クロックドメイン間の通信
- CDC (Clock Domain Crossing) 問題
- タイミング制約が複雑
- デバッグが困難

✓ 単層同期回路方式（採用）

- 全モジュールが同じCLK (125MHz)
- ENABLE信号で制御（assign文で生成）
- CARRY信号も組み合わせ回路
- 全カウンタが同じposedge CLKで更新
- 同期クロックドメイン内で完結
- タイミング制約がシンプル
- デバッグが容易
- 高速動作が可能

【設計の要点】

- ✓ 論理的な依存関係(Carry連鎖)と物理的な同期を分離
- ✓ Carryは組み合わせ回路で伝播(always @内の条件)
- ✓ 更新は全て同じposedge CLKで実行
- ✓ ENABLEによる統一的な制御

9-1-4. なぜ同期できるのか：FPGAのタイミング特性

【FPGAの基本動作】

- ① クロックの立ち上がりで値を取り込む
- ② 取り込んだ値で組み合わせ回路が動作
- ③ 次のクロックまでに結果が確定
- ④ 次のクロックの立ち上がりで新しい値を取り込む

【重要な特性】

CLKの立ち上がり時点では「前の値」として認識される

例：tmp_count が 124999999 → 0 に変わる瞬間

| | |
|---------------------------------------|--|
| CLK立ち上がり時点： | |
| tmp_count = 124999999 と認識 | |
| ENABLE = (tmp_count == 124999999) = 1 | |
| | |
| CLK立ち上がり後(数ns後)： | |
| tmp_count = 0 に更新 | |
| ENABLE = 0 に変化(次のCLKで認識) | |

→ 「124999999の最後の瞬間」にENABLE=1として判定可能

→ この特性により、秒・分・時が同時に条件判定できる

9-1-5. 同時判定の仕組み: 23:59:59 → 00:00:00 の例

【CLKの立ち上がり時点での状態】

tmp_count = 124999999 (ENABLEが立つ最後の瞬間)

【すべて同じCLKエッジで同時に判定完了】

→ 次のCLKエッジで全カウンタが 00:00:00 に更新

→ 時間的な遅延なし、完全に同期

秒カウンタ(CNT60):

CNT10 = 9, CNT6 = 5 (59のまま)

CARRY_in = 1 (常に1)

→ CARRY = (9==9) && 1 = 1 ← 組み合わせ回路で即座に確定

→ CARRY_out = (5==5) && 1 = 1

分カウンタ(CNT60):

CNT10 = 9, CNT6 = 5 (59のまま)

CARRY_in = 秒のCARRY_out = 1

→ CARRY = (9==9) && 1 = 1

→ CARRY_out = (5==5) && 1 = 1

時カウンタ(CNT24):

CNT10 = 3, CNT3 = 2 (23のまま)

CARRY_in = 分のCARRY_out = 1

→ CARRY = ({2,3}==23) && 1 = 1

→ CARRY_out = (2==2) && 1 = 1

9-1-6. 実装コード: 同期の仕組み

【CNT60.vの実装(抜粋)】

// ①内部CARRY信号の生成(組み合わせ回路)

```
always @(CNT10 or CARRY_in) begin
    if (CNT10 == 4'h9 && CARRY_in == 1'b1)
        CARRY <= 1'b1; // 数ns以内に確定
    else
        CARRY <= 1'b0;
end
```

// ②CARRY_out信号の生成(組み合わせ回路)

```
always @(CNT6 or CARRY) begin
    if (CNT6 == 4'h5 && CARRY == 1'b1)
        CARRY_out <= 1'b1; // CARRYが確定後、すぐ確定
    else
        CARRY_out <= 1'b0;
end
```

// ③カウント更新(同期回路)

```
always @(posedge CLK or posedge RESET) begin
    if (RESET == 1'b1)
        CNT10 <= 4'h0;
    else if (ENABLE == 1'b1 && CARRY_in == 1'b1)
        // ↑この条件判定が全カウンタで同時に行われる
        if (CARRY == 1'b1)
            CNT10 <= 4'h0;
        else
            CNT10 <= CNT10 + 4'h1;
end
```

【ポイント】

- ・CARRY、CARRY_outは組み合わせ回路(always @(*))
 - ・更新判定はすべて同じCLKエッジ(always @(posedge CLK))
- 秒・分・時が完全同期

9-1-7. 実装ソースコード（単層同期回路）

```
// SEC1.v: ENABLE信号生成（組み合わせ回路）
assign ENABLE = (tmp_count == (SEC1_MAX - 1)) ? 1'b1 : 1'b0;

// CLOCK_ALL.v: 全カウンタが同じCLKとENABLEで動作
CNT60 i0(.CLK(CLK), .RESET(RESET), .ENABLE(ENABLE),
        .CARRY_in(1'b1), .CARRY_out(CARRY), ...);

CNT60 i2(.CLK(CLK), .RESET(RESET), .ENABLE(ENABLE),
        .CARRY_in(CARRY), .CARRY_out(CARRY_2), ...);

CNT24 i4(.CLK(CLK), .RESET(RESET), .ENABLE(ENABLE),
        .CARRY_in(CARRY_2), .CARRY_out(CARRY_3), ...);

// CNT60.v: CARRY信号も組み合わせ回路
always @(CNT10 or CARRY_in) begin
    if (CNT10 == 4'h9 && CARRY_in == 1'b1)
        CARRY <= 1'b1;
    else
        CARRY <= 1'b0;
end

// カウント更新は同じposedge CLKで実行
always @(posedge CLK or posedge RESET) begin
    if (ENABLE == 1'b1 && CARRY_in == 1'b1) begin
        // カウント更新処理
    end
end
end
```

9-2. 技術的工夫②：ブロックRAMによる曜日計算最適化

【課題】

ツェラーの公式で曜日を計算すると複雑な演算（乗算・除算・剰余）が必要
組み合わせ回路で実装すると Critical Path（最長遅延パス）になり、動作周波数が低下

【解決策】

- ① C言語（weekday.c）で2000年～2100年の各月1日の曜日を事前計算
- ② weekday_coe.c でブロックRAM用の.coeファイルを生成（1212エントリ）
- ③ Vivado IP Catalog でブロックRAMを生成、.coeファイルを初期値として設定
- ④ weekday_calc.v でブロックRAMを使用して曜日を高速ルックアップ
- ⑤ 各月1日の曜日 + (日 - 1) を7で割った余りで任意の日の曜日を計算
- ⑥ 複雑な演算を「メモリ参照 + 簡単な加算」に置き換え → 遅延を劇的に改善

【効果】演算回路を削減し、Critical Path を短縮 → 高速動作を実現

9-3. 技術的工夫③：ダイナミック点灯制御の最適化

【課題】

4桁7セグメントLEDは同時に全桁表示できない構造

→ 1桁ずつ高速に切り替えて点灯（ダイナミック点灯）が必要

初期実装：デコーダ4個使用、32bitデータ幅 → 回路規模が大きい

【改善策】

- ① デコーダを4個 → 1個に削減
 - ・桁選択カウンタ（2bit）で現在表示する桁を選択
 - ・選択された桁のBCDデータ（4bit）だけをデコーダに入力
 - ・SA[3:0]信号で対応する桁のコモン端子を有効化
- ② データ幅を32bit → 16bitに削減
 - ・4桁分のBCDデータ（各4bit）を16bitで表現
 - ・必要なデータだけを抽出して処理
- ③ 高速クロック（数kHz）で桁を切り替え → 人間の目には同時点灯に見える

【効果】リソース使用量を75%削減、回路をシンプル化

【4桁を高速切替：デコーダ1個、データ幅16bit】

```
module DCOUNT(CLK, ENABLE, L1, L2, L3, L4, SA, L);
input CLK, ENABLE;
input [3:0] L1, L2, L3, L4;
output [3:0] SA;
output [3:0] L;

parameter MAX_COUNT = 3'b111;
reg [2:0] sa_count_tmp;
reg [3:0] sa_count;
reg [3:0] L_tmp;

assign SA[3] = (sa_count[3]==1'b1)? 1'b1 : 1'b0;
assign SA[2] = (sa_count[2]==1'b1)? 1'b1 : 1'b0;
assign SA[1] = (sa_count[1]==1'b1)? 1'b1 : 1'b0;
assign SA[0] = (sa_count[0]==1'b1)? 1'b1 : 1'b0;
assign L = L_tmp;

always @(posedge CLK)
begin
    if (ENABLE==1'b1)
        if (sa_count_tmp==MAX_COUNT)
            sa_count_tmp <= 3'b000;
        else
            sa_count_tmp <= sa_count_tmp + 1'b1;
end

always @(posedge CLK)
begin
    if (sa_count_tmp[0]==1'b0)
        begin
            sa_count <= 4'b0000; L_tmp <= L_tmp;
        end
    else
        case (sa_count_tmp[2:1])
            2'b00:begin
```

【4桁を高速切替：デコーダ1個、データ幅16bit】

```
        sa_count <= 4'b1000;L_tmp <= L4;
    end
    2'b01:begin
        sa_count <= 4'b0100;L_tmp <= L3;
    end
    2'b10:begin
        sa_count <= 4'b0010;L_tmp <= L2;
    end
    2'b11:begin
        sa_count <= 4'b0001;L_tmp <= L1;
    end
    default:begin
        sa_count <= 4'bxxxx;L_tmp <= 8'bxxxxxxxx;
    end
endcase
end

endmodule
```

9-4. 技術的工夫④：MSEモジュールの統合設計

【MSE = Metastability + Switch(Chattering) + Edge】

【3つの機能を1モジュールに統合】

- ① メタステビリティ対策
 - ・外部入力を2段のフリップフロップで受けて安定化（FF同期化）
 - ・非同期信号 → 同期信号への変換で、不定状態を防ぐ
- ② チャタリング対策
 - ・ボタン押下時の電氣的ノイズ（バウンス）を除去
 - ・ENABLE_kHz信号（数kHz）でサンプリングレートを制限
 - ・高速なノイズは無視され、安定した信号だけが通過
- ③ エッジ検出
 - ・立ち上がりエッジ（0→1の瞬間）を検出してパルス出力
 - ・ボタンを押しっぱなしにしても1回だけカウント
 - ・前回の状態を保持し、現在との差分でエッジを判定

【効果】3つの処理を1モジュールに集約 → コード量削減、再利用性向上

9-5. 技術的工夫⑤：レジスタの統一

【課題】

時刻表示用レジスタと設定用レジスタが別々に存在

→ レジスタ数が2倍（年月日時分秒で各2系統 = 合計12個）

→ 設定完了時に表示用レジスタへのコピー処理が必要

→ 回路規模の増大、処理の複雑化

【改善策】

① レジスタを1系統に統一（年月日時分秒で各1個 = 合計6個）

② 通常動作時：ENABLE信号により時刻が自動的に進む

③ 設定モード時：該当カウンタのLOAD信号を有効化し、設定中の桁だけを変更
他の桁は通常通りカウント動作を継続

④ 設定完了後：すぐに通常動作に戻る
→ コピー処理が不要、シームレスな動作

【効果】レジスタ数を半減、処理をシンプル化 → リソース削減50%

9-6. 技術的工夫⑥：境界値テストによる厳密な検証

【境界値テスト = 値が切り替わる瞬間の動作を確認】

- ① 2000/02/28 23:59:59 → 2000/02/29 00:00:00
(うるう年の2月末 → 2月29日への遷移)
- ② 2000/02/29 23:59:59 → 2000/03/01 00:00:00
(うるう年の2月29日 → 3月への遷移、曜日も確認)
- ③ 2001/02/28 23:59:59 → 2001/03/01 00:00:00
(平年の2月末 → 3月への遷移、2月29日がスキップされることを確認)
- ④ 2000/12/31 23:59:59 → 2001/01/01 00:00:00
(年またぎ、月・日・時分秒すべてがリセット)
- ⑤ 2099/12/31 23:59:59 → 2100/01/01 00:00:00
(世紀またぎ、2100年はうるう年ではないことを確認)
- ⑥ 2100/12/31 23:59:59 → 2000/01/01 00:00:00
(101年のロールオーバー、正しく2000年に戻ることを確認)

【結果】すべての境界値で正しく動作 → 実装の正確性を証明

9-7. 技術的工夫⑦ : TEST_UPDOWN10.v (7セグLED表示)

【開発初期の検証手法】

【課題】

- ・実機 (FPGAボード) でテストする前に、動作を確認したい
- ・LED[7:0]の8bit信号だけでは、何が表示されるか分かりにくい
- ・例 : LED = 8'b00111000 → これが何の数字/文字なのか即座に判断できない

【解決策】

- ・テストベンチで7セグLEDの「形」をTCL Consoleに表示
- ・LED[7:0]の各ビットをセグメント(A-G, Dp)に対応させて視覚化
- ・シミュレーション中に意図通りの表示になっているか目視確認

【実装の仕組み】

- ① LED[7:0]の各ビットに応じて文字列を生成
LED[7] (A) → "----" または " " (空白)
LED[6] (B), LED[5] (C), LED[3] (E), LED[2] (F) → "| " または " "
LED[4] (D), LED[1] (G) → "----" または " "
LED[0] (Dp) → " ." または " "
- ② \$display()で組み合わせて7セグLEDの形を出力
- ③ シミュレーション実行中、リアルタイムで表示が更新される

9-7. 技術的工夫⑦ : TCL Consoleでの表示例 (2/3)

【TCL Consoleに7セグLEDを描画】

```
module TEST_UPDOWN10;
    reg [8*4:1] A_DISP, D_DISP, G_DISP;          // 横線セグメント (4文字分)
    reg [8*2:1] B_DISP, C_DISP, E_DISP, F_DISP, Dp_DISP; // 縦線+小数点 (2文字分)
    parameter TURN_ON = 1'b1;

    // LEDが変化したら表示を更新
    always @(LED) begin
        for (j = 7; j >= 0; j = j - 1) begin
            case (j)
                7: if (LED[j] === TURN_ON) A_DISP = "----"; // セグメントA
                   else A_DISP = " ";
                6: if (LED[j] === TURN_ON) B_DISP = "| "; // セグメントB
                   else B_DISP = " ";
                5: if (LED[j] === TURN_ON) C_DISP = "| "; // セグメントC
                   else C_DISP = " ";
                4: if (LED[j] === TURN_ON) D_DISP = "----"; // セグメントD
                   else D_DISP = " ";
                3: if (LED[j] === TURN_ON) E_DISP = "| "; // セグメントE
                   else E_DISP = " ";
                2: if (LED[j] === TURN_ON) F_DISP = "| "; // セグメントF
                   else F_DISP = " ";
                1: if (LED[j] === TURN_ON) G_DISP = "----"; // セグメントG
                   else G_DISP = " ";
                0: if (LED[j] === TURN_ON) Dp_DISP = " ."; // 小数点
                   else Dp_DISP = " ";
            endcase
        end

        #5 // 少し遅延させて表示
        $display("");
        $display(" %s",A_DISP); // ---- (A)
        $display(" %s %s",F_DISP, B_DISP); // | | (F, B)
        $display(" %s",G_DISP); // ---- (G)
        $display(" %s %s",E_DISP, C_DISP); // | | (E, C)
        $display(" %s %s %s",D_DISP, Dp_DISP); // ---- . (D, Dp)
        $display("");
    end
endmodule
```

9-7. 技術的工夫⑦:TCL Consoleでの表示例(3/3)

【シミュレーション実行時の出力】

数字「0」の表示例

```
LED = 8'b111111100
```

```
---- (A点灯)
|  | (F, B点灯)
    (G消灯)
|  | (E, C点灯)
---- (D点灯, Dp消灯)
```

→ 0の形になる

数字「8」の表示例

```
LED = 8'b111111110
```

```
---- (A点灯)
|  | (F, B点灯)
---- (G点灯)
|  | (E, C点灯)
---- (D点灯, Dp消灯)
```

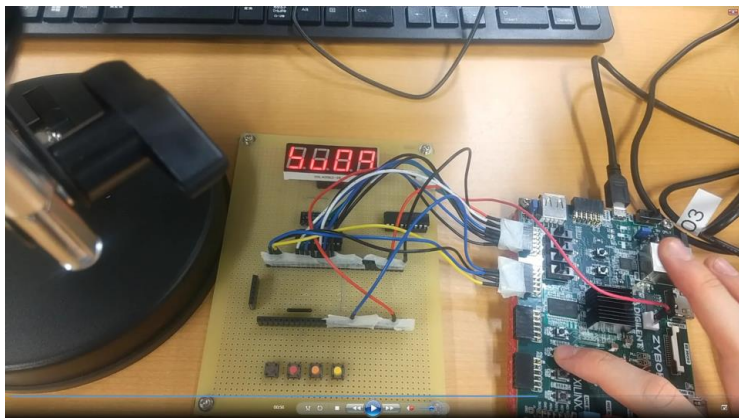
→ 8の形になる

【効果】

- ✓ 実機テスト前にシミュレーションで動作確認できる
- ✓ デバッグ時間を大幅短縮（バグを早期発見）
- ✓ 意図しない表示を即座に発見できる
- ✓ FPGAへの書き込み回数を削減（開発効率向上）

9-8. 技術的工夫⑧：オリジナル曜日表示パターン（1/2）

【7セグメント表示形状】ビット順: ABCDEFG_Dp





実機表示例：SU（日曜日）

| | | | | | |
|-------|------|-------|------|-------|------|
| SU-S | SU-U | MO-M | MO-O | TU-T | TU-U |
| | | | | | |
| 日曜 SU | | 月曜 MO | | 火曜 TU | |
| | | | | | |
| WE-W | WE-E | | | | |
| | | | | | |
| 水曜 WE | | | | | |

9-9. 技術的工夫⑧:オリジナル曜日表示パターン(2/2)

【残りの曜日表示パターン】

TH-T TH-H

木曜 TH

FR-F FR-R

金曜 FR

SA-S SA-A

土曜 SA

【設計思想】

- ・ 7セグメントで表現可能な2文字のアルファベットで各曜日を表現
- ・ 全7曜日でユニークなパターンを実現
- ・ 可読性を重視したデザイン

ビット順序: ABCDEFG_Dp

A=上横, B=右上縦, C=右下縦, D=下横, E=左下縦, F=左上縦, G=中横, Dp=小数点

9-10. 技術的工夫⑧：曜日表示の設計思想（1/2）

【2桁目・1桁目を個別制御】

```
// DECODER7.vの曜日表示部分（抜粋）
always @(COUNT or TOP_CURRENT_STATE or DIS_CURRENT_STATE or SA) begin
    if(TOP_CURRENT_STATE == 1'b1 && DIS_CURRENT_STATE[0] == 1'b1 && SA[3:2] == 2'b00) begin
        case(SA[1:0])
            2'b01: // 1桁目 (U, O, U, E, H, R, A)
                case(COUNT)
                    4'b0000: LED <= 8'b00111100_0; // U (日曜日)
                    4'b0001: LED <= 8'b00111101_0; // O (月曜日)
                    4'b0010: LED <= 8'b00111100_0; // U (火曜日)
                    4'b0011: LED <= 8'b10011111_0; // E (水曜日)
                    4'b0100: LED <= 8'b00101111_0; // H (木曜日)
                    4'b0101: LED <= 8'b00001011_0; // R (金曜日)
                    4'b0110: LED <= 8'b11110111_0; // A (土曜日)
                    default: LED <= 8'b00111100_0;
                endcase

            2'b10: // 2桁目 (S, M, T, W, T, F, S)
                case(COUNT)
                    4'b0000: LED <= 8'b00111011_0; // S (日曜日)
                    4'b0001: LED <= 8'b11110110_0; // M (月曜日)
                    4'b0010: LED <= 8'b00011111_0; // T (火曜日)
                    4'b0011: LED <= 8'b01010101_0; // W (水曜日)
                    4'b0100: LED <= 8'b00011111_0; // T (木曜日)
                    4'b0101: LED <= 8'b10001111_0; // F (金曜日)
                    4'b0110: LED <= 8'b00111011_0; // S (土曜日)
                    default: LED <= 8'b00111011_0;
                endcase

            endcase
        endcase
    end
    // ... 他の表示モード ...
end
```

// SA[3:2]で桁選択、SA[1:0]で1桁目/2桁目を選択
// COUNT[3:0]で曜日番号 (0=日, 1=月, ..., 6=土)

9-10. 技術的工夫⑧:曜日表示の設計思想 (2/2)

【制約条件と設計の工夫】

【制約条件】

- ・7セグメントLEDは数字表示用の形状
- ・アルファベットを表示するには限界がある
- ・一部の文字は複数の解釈が可能 (例 : 5とS、0とO)

【設計の工夫】

- ① 2文字の組み合わせで明確化
 - 単独では曖昧でも、2文字セットで判別可能
 - 例 : 「S」単独では5か? → 「SU」なら日曜日と分かる
- ② 各曜日で異なる文字の組み合わせ
 - 7種類すべてがユニークな組み合わせ
 - 誤認識を防ぐ
- ③ 英語の曜日名を短縮
 - Sunday → SU, Monday → MO, Tuesday → TU, ...
 - 直感的に理解しやすい
- ④ 7セグメントで表現可能な文字を選択
 - A, E, F, H, M, O, R, S, T, U, W が使用可能
 - これらを組み合わせで7種類を実現

【成果】

- ✓ 限られたリソースで曜日表示を実現
- ✓ オリジナルのパターン設計により、視認性を確保
- ✓ ダイナミック点灯との組み合わせで、2桁表示を実現

9-11. ワンホットエンコーディングの採用

【バイナリエンコーディング vs ワンホットエンコーディング】

✕ バイナリ方式（一般的な方法）

- 7状態を3ビットで表現 (000, 001, 010, ..., 110)
- 状態判定: if (CURRENT_STATE == 3'b010) → 3ビット全体を比較
- デコード回路が必要 → LUT使用量増加
- 全ビットを各モジュールに配線

✓ ワンホット方式（採用した方式）

- 7状態を7ビットで表現 (0000001, 0000010, 0000100, ...)
- 状態判定: if (SET_CURRENT_STATE[0] == 1'b1) → 1ビットのみチェック
- デコード回路不要 → LUT削減
- 必要なビットだけを各モジュールに配線

【実装例】

parameter L1 = 7'b0000001, L2 = 7'b0000010, L3 = 7'b0000100, ...

// CNT60.vでは7ビット中2ビットだけ受け取る

input [1:0] SET_CURRENT_STATE;

else if (((ENABLE && CARRY_in) && SET_CURRENT_STATE[0] == 1'b1) ||

(SET_CURRENT_STATE[1] == 1'b1 && INC_MODE == 1'b1))

// ↑ 単一ビットチェックで高速判定！

【設計上の利点】

- ✓ 配線削減: 必要なビットのみ配線
- ✓ 高速判定: デコード回路不要、クリティカルパス短縮
- ✓ デバッグ容易: 波形で状態が一目瞭然
- ✓ 拡張性: 状態追加時にビット幅+1のみ

9-12. シンセシス結果：ハードウェアリソース使用量

【Vivado シンセシス結果】

ターゲットデバイス: Artix-7 (XC7A35TICSG324-1L)

| リソース種類 | 使用量 | 利用可能 | 使用率 |
|----------------------|------|---------|-------|
| Slice LUTs | 390 | 53,200 | 0.73% |
| └ LUT as Logic | 390 | | |
| └ LUT as Memory | 0 | | |
| Slice Registers (FF) | 154 | 106,400 | 0.14% |
| └ カウンタ用 | ~130 | | |
| └ 状態レジスタ用 | ~24 | | |
| F7 Muxes | 10 | 26,600 | 0.04% |
| Block RAM (BRAM) | 0.5 | 50 | 1.00% |
| └ 曜日計算用 (RAMB18E1) | 1個 | (半分使用) | |
| DSP | 0 | 90 | 0.00% |
| Bonded IOB | 17 | 210 | 8.10% |

【ワンホットエンコーディングの影響】

- Slice Registers: +24個 (DIS_MODE: 7bit, SET_MODE: 7bit, MAIN_MODE: 5bit等)
- LUT削減: 状態デコード回路が不要 → 約10-15 LUT削減
- 配線リソース削減: 必要なビットのみ配線

【最大動作周波数】

- 達成周波数: 125 MHz以上 (設計目標: 125MHz)
- クリティカルパス: tmp_count → ENABLE → カウンタ更新

【設計効率】

- ✓ 小規模FPGA (Artix-7) で1%未満のリソース使用
- ✓ 拡張の余地が十分にある
- ✓ 単層同期設計により安定した高速動作を実現

10-1. 的当てゲーム製作(技能祭展示作品) (1/2)

【技能祭での展示風景（2024年10月）】



10-1. 的当てゲーム製作(技能祭展示作品) (2/2)

【システム概要】

FPGA (Basys3)

- ・32×32 LEDマトリクス制御
- ・的とボールの表示
- ・リアルタイム描画

ラズベリーパイ

- ・カメラで画像取得
- ・OpenCV (C++) で処理
- ・ゲームロジック実装
- ・当たり判定計算

【ゲームの流れ】

- ① プレイヤーがボールを投げる
- ② カメラでボール軌跡を撮影
- ③ ラズパイで画像処理
 - ・色検出
 - ・座標抽出
 - ・軌跡予測
- ④ 当たり判定
- ⑤ FPGAに結果送信
- ⑥ LEDマトリクスに表示

【開発の特徴】

Claude AIを活用して効率的に開発

複雑な画像処理ロジックやFPGA制御コードの実装をAIとの対話で段階的に改善
エラーが出たら原因を特定し、修正案を提示してもらうサイクルで高速開発

10-2. 的当てゲーム：技術的工夫

【独自アイデア：X座標転換点による当たり判定】

- ① ボールの軌跡をカメラで連続撮影し、座標データを記録
- ② 座標データをExcelでグラフ化して視覚的に分析
- ③ X座標の変化を観察すると、的の位置付近で「転換点」を発見
→ X座標が増加から減少（または減少から増加）に転じる瞬間
- ④ この転換点で的の座標範囲内にボールがあれば「当たり」と判定
- ⑤ 従来の距離計算（ $\sqrt{(x^2+y^2)}$ ）より高精度かつ計算コストが低い

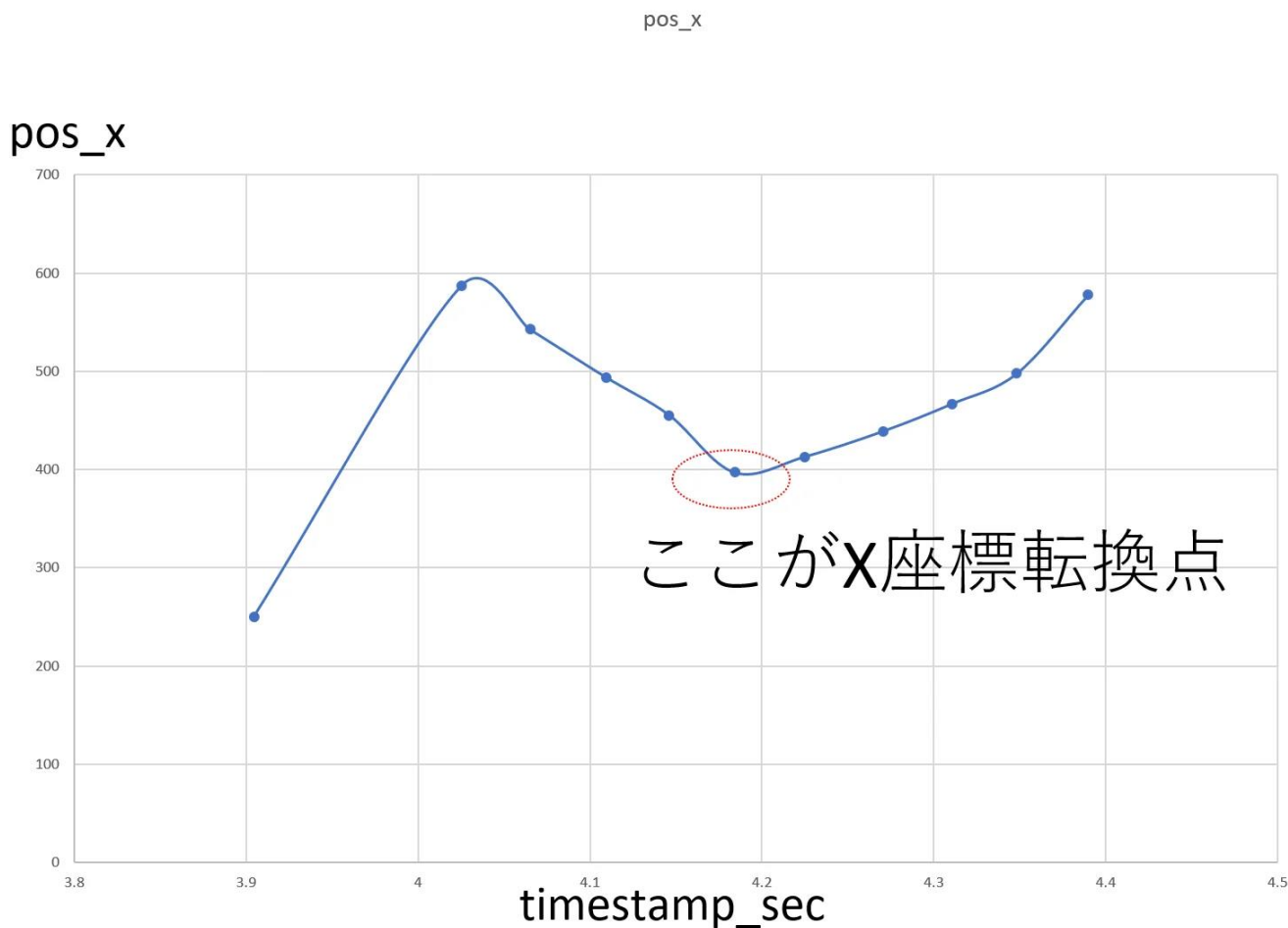
【FPGA + ラズパイのハイブリッド構成】

FPGA：高速なLED制御、リアルタイム描画に特化
ラズパイ：複雑な画像処理、ゲームロジックを柔軟に実装
SPI/UARTで通信し、互いの強みを活かしたシステム構築
→ 開発の効率化、機能拡張の容易さを実現

10-3. 的当てゲーム：X座標転換点による当たり判定

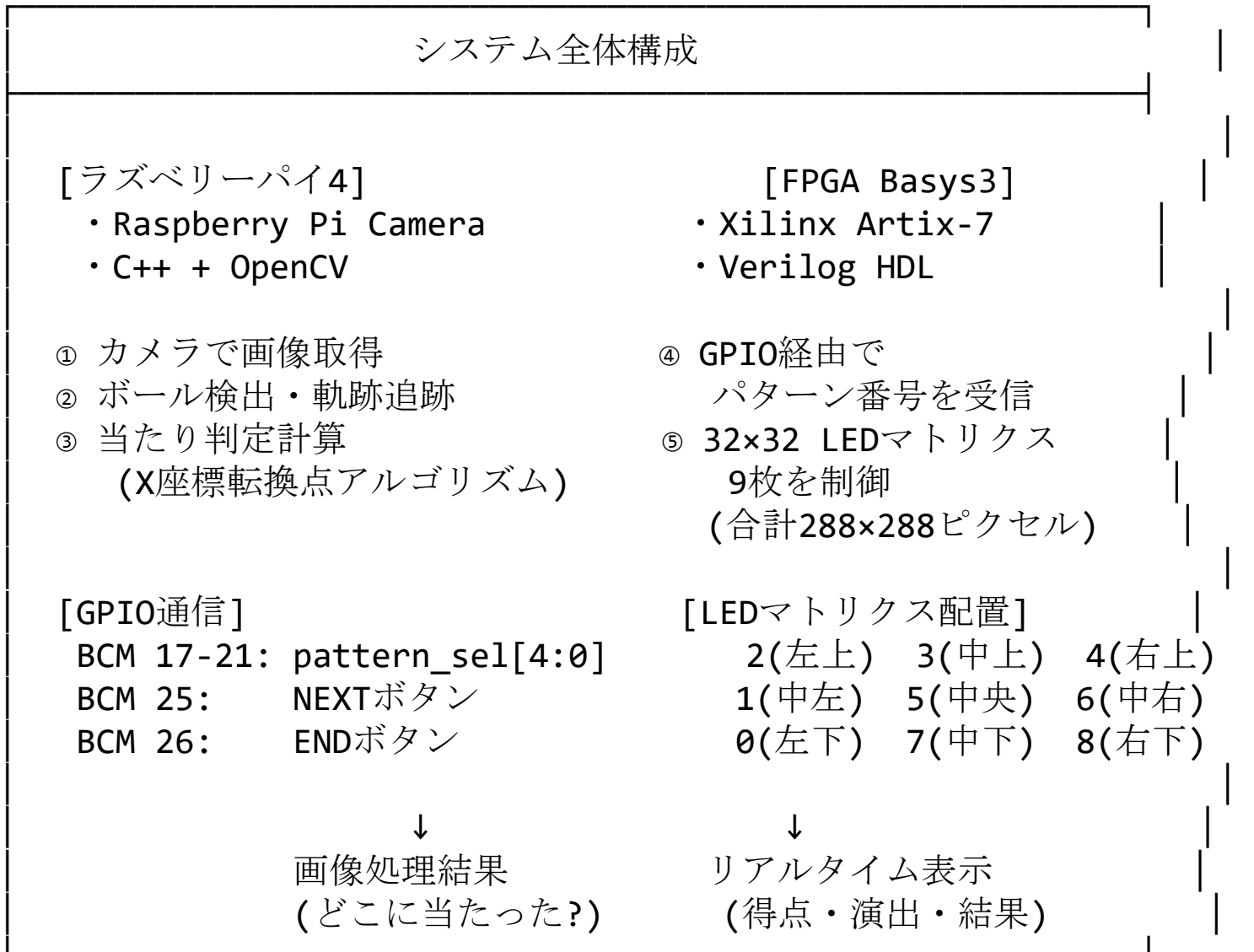
【独自アイデア：Excelでボール軌跡をグラフ化→X座標の転換点で当たり判定】

ボールのX座標をExcelでグラフ化し、X座標が変化する転換点（山や谷）で当たり判定を行う独自のアルゴリズムを考案。シンプルかつ効果的な判定を実現。



10-3. 的当てゲーム：システム全体構成

【ハードウェア構成】



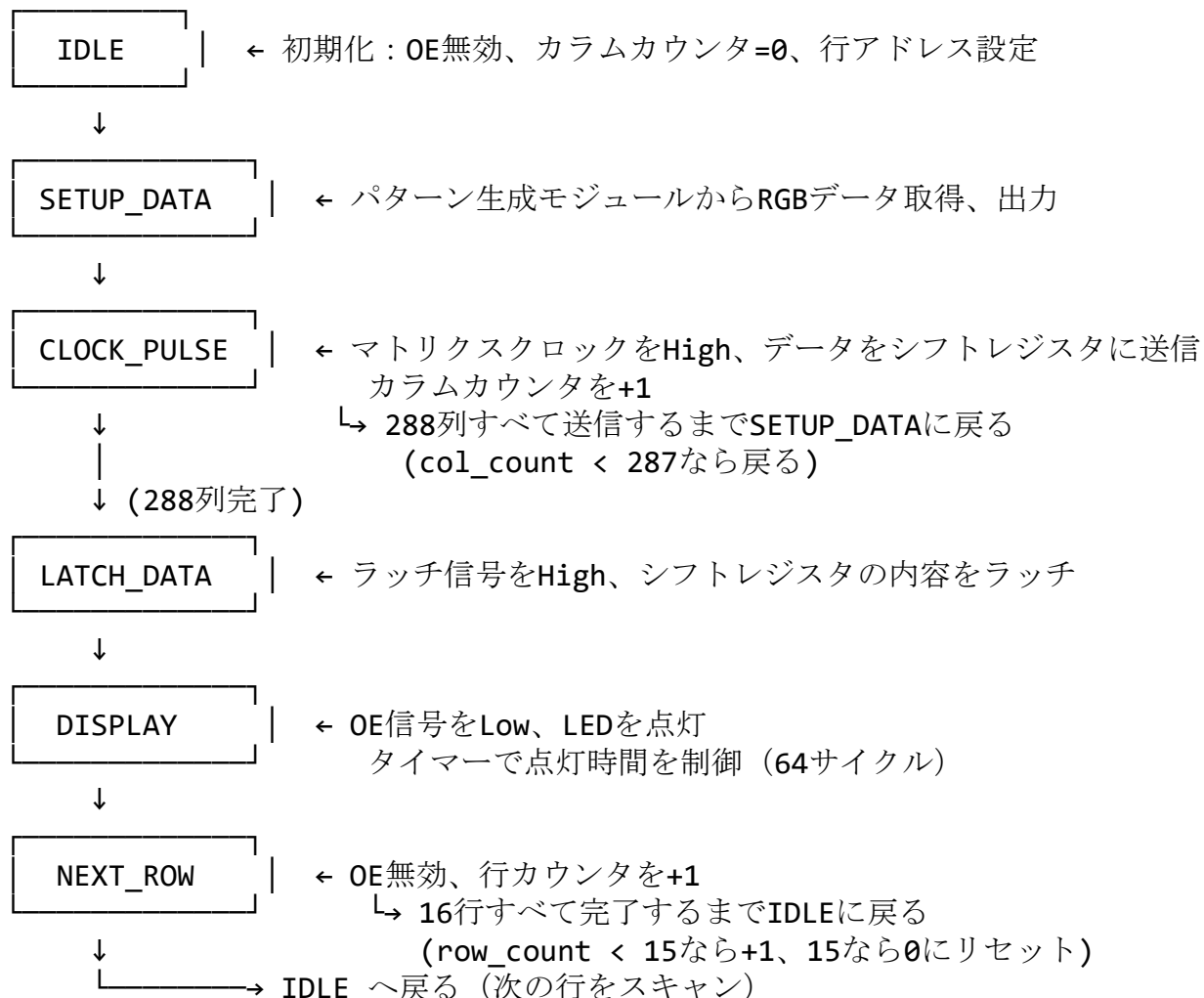
10-4. 的当てゲーム：FPGAモジュール階層

【3つの主要モジュール】

```
rgb_matrix_top (トップモジュール)
├── clock_divider (クロック分周器)
│   └── 12MHz → 約300kHz に分周 (clk_enable信号生成)
├── state_controller (状態遷移マシン)
│   ├── 6つの状態で LEDマトリクスを制御
│   ├── IDLE → SETUP_DATA → CLOCK_PULSE → LATCH_DATA → DISPLAY → NEXT_ROW
│   ├── 288列 (32×9) × 16行 = 4,608ピクセルを順次スキャン
│   └── RGB信号、アドレス、ラッチ、クロック、OE信号を生成
└── pattern_generator (パターン生成モジュール)
    ├── pattern_sel[4:0]で32種類のパターンを切替
    ├── 27種類のゲーム画面を実装：
    │   ├── - STANDBY (0): "START"文字表示
    │   ├── - EXPLANATION (1): 各パネルに得点数字表示 (10点/3点/5点)
    │   ├── - GAME_PLAYING (2): ゲーム中 (枠のみ)
    │   ├── - HIT演出 (3-11): 各パネルが赤く点灯
    │   ├── - MISS (12): 全画面赤
    │   └── - RESULT (13-28): 中央に得点を大きく表示 (0-30点)
    └── 7セグメント数字フォント、アルファベットフォント内蔵
```

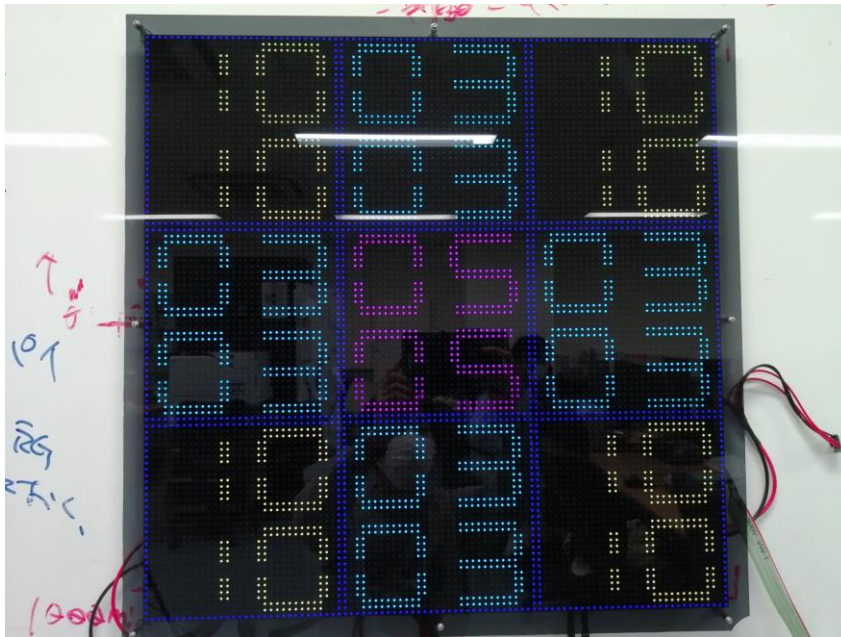

10-5. 的当てゲーム：状態遷移マシン

【6状態で LEDマトリクスをスキャン】

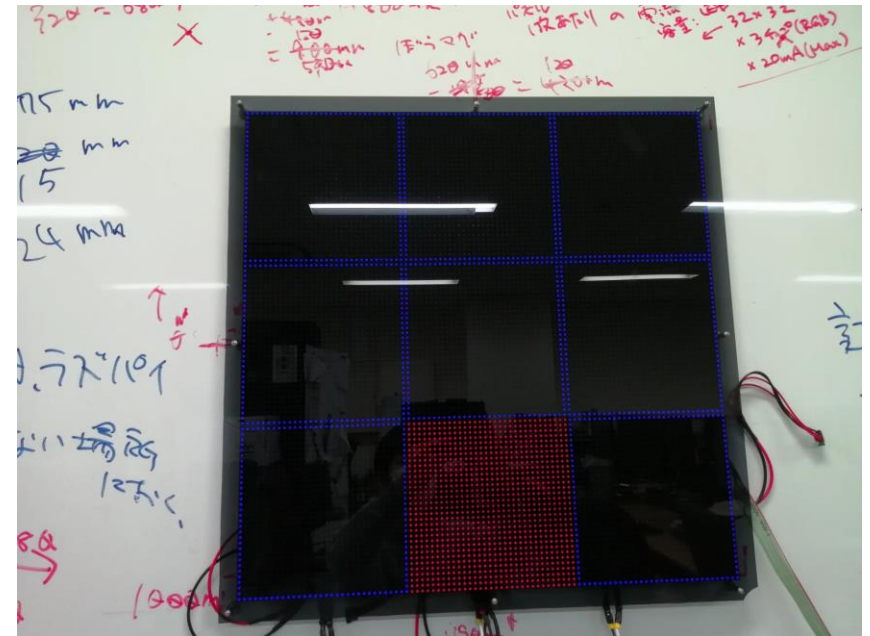


【1フレーム = 16行 × (288列 + 処理時間) ≈ 60Hz でリフレッシュ】

10-7. 的当てゲーム：ラズパイ側の制御（解説）



得点表示パターン
各パネルに10点/3点/5点を表示



ゲーム中パターン
青枠+下部赤パネル点灯

10-8. 的当てゲーム : state_controller.v (抜粋)

【ラズパイ側の役割】

1. ゲーム全体の進行管理
 - 状態遷移（待機→説明→ゲーム→結果）
 - タイマー管理
 - 得点計算
2. FPGAへの指示送信
 - SPI通信でパターンデータを送信
 - 状態に応じた表示パターンを指定
3. センサー入力の処理
 - ボールの着弾検知
 - どのパネルに当たったかを判定
 - 得点を加算
4. 音声・効果音の再生
 - ゲーム開始音
 - 着弾音
 - 終了音

10-9. 的当てゲーム : pattern_generator.v の仕組み

【状態遷移の実装】

```
// 状態定義
localparam [2:0]
    IDLE = 3'b000,
    SETUP_DATA = 3'b001,
    CLOCK_PULSE = 3'b010,
    LATCH_DATA = 3'b011,
    DISPLAY = 3'b100,
    NEXT_ROW = 3'b101;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        col_count <= 9'd0;
        row_count <= 4'd0;
        // ... 初期化 ...
    end else if (clk_enable) begin
        case (state)
            IDLE: begin
                oe_n <= 1'b1;           // OE無効 (LED消灯)
                lat <= 1'b0;
                clk_out <= 1'b0;
                addr <= row_count;       // 行アドレス設定
                col_count <= 9'd0;
                state <= SETUP_DATA;
            end

            SETUP_DATA: begin
                clk_out <= 1'b0;
                // pattern_generatorからRGBデータを取得
                r1 <= upper_r_in; g1 <= upper_g_in; b1 <= upper_b_in;
                r2 <= lower_r_in; g2 <= lower_g_in; b2 <= lower_b_in;
                state <= CLOCK_PULSE;
            end
        endcase
    end
end
```

```
CLOCK_PULSE: begin
    clk_out <= 1'b1;           //
    マトリクスクロックをHigh
    if (col_count < 287) begin
        col_count <= col_count
        + 1;
        state <= SETUP_DATA;
    // 次の列へ
    end else begin
        state <= LATCH_DATA;
    // 全列完了
    end
end
// LATCH_DATA, DISPLAY,
NEXT_ROW は省略
endcase
end
end
```

10-10. 的当てゲーム : pattern_generator.v (抜粋)

【27種類のパターン生成】

【入力】

- pattern_sel[4:0]: ラズパイから受信するパターン番号 (0-31)
- col_count[8:0]: 現在の列位置 (0-287)
- row_count[3:0]: 現在の行位置 (0-15)

【出力】

- upper_r, upper_g, upper_b: 上半分のRGB (1bit各)
- lower_r, lower_g, lower_b: 下半分のRGB (1bit各)

【パターン例】

- pattern_sel = 0: STANDBY画面
 - 中央パネル (panel_num=5) に "START" 文字を表示
 - 5×7ビットマップフォントで描画
- pattern_sel = 1: EXPLANATION画面
 - 各パネルに得点数字 (10点/3点/5点) を7セグメント風に表示
 - get_panel_score()関数でパネルごとの得点を判定
- pattern_sel = 3-11: HIT演出
 - 当たったパネルだけを赤く点灯
- pattern_sel = 13-28: RESULT画面
 - 中央に得点を大きく表示 (0-30点)
 - 16×16ピクセルの7セグメント数字フォント

10-11. 的当てゲーム：ラズパイ側の制御

【パネル番号の計算とパターン生成】

```
// パネル番号計算 (0-8)
wire [3:0] panel_num = col_count[8:5]; // 288列 ÷ 32 = 9パネル

// 各パネル内での位置 (0-31)
wire [4:0] local_col = col_count[4:0];

// 得点判定関数
function [3:0] get_panel_score;
    input [3:0] panel;
    begin
        case (panel)
            4'd0, 4'd2, 4'd4, 4'd8: get_panel_score = 4'd10; // 4隅：10点
            4'd1, 4'd3, 4'd6, 4'd7: get_panel_score = 4'd3;  // 4辺：3点
            4'd5: get_panel_score = 4'd5;                    // 中央：5点
            default: get_panel_score = 4'd0;
        endcase
    end
endfunction
```

```

// パターン別処理
always @(*) begin
    case (pattern_sel)
        5'd0: begin // STANDBY
            if (panel_num == 4'd5) begin // 中央パネル
                // "START"文字を表示
                char_pos = local_col - text_start;
                show_text = (char_pos >= 0 && char_pos <
30 &&
                                row_count >= text_row_start &&
                                row_count < text_row_start +
7);
                if (show_text) begin
                    // 5文字目 "T", 4文字目 "R", ... を判定し
て描画
                    // get_char_pixel()関数で5×7ビットマップか
ら点灯判定
                    upper_r = pixel_show; // 緑色で表示
                    upper_g = pixel_show;
                    upper_b = 1'b0;
                end
            end
        end
        5'd1: begin // EXPLANATION
            score = get_panel_score(panel_num);
            // 7セグメント数字を描画
            pixel_show = get_7seg_pixel(tens_digit,
local_col, row_count);
            // 得点に応じて色分け(10点=黄、3点=シアン、5点=マゼン
タ)
        end
        // ... 他のパターンも同様 ...
    endcase
end

```

10-12. 的当てゲーム：開発の工夫と成果

【C++ + OpenCV + GPIO制御】

【target_game.cpp の主な機能】

- ① GPIO初期化
 - ・BCM 17-21をOUTPUTモードに設定 (pattern_sel[4:0])
 - ・BCM 25, 26をINPUTモード (NEXTボタン、ENDボタン)
- ② ゲームループ
 - ・状態遷移：STANDBY → EXPLANATION → GAME_PLAYING → RESULT
 - ・各状態でpattern_selを変更してFPGAに送信
- ③ ball_detector_4との連携
 - ・カメラ画像処理プロセスから当たり判定結果を受信
 - ・/tmp/hit_result.txt ファイルでパネル番号 (0-8) を取得
 - ・該当するHIT演出パターン (3-11) をFPGAに送信
- ④ 得点計算
 - ・当たったパネルに応じて得点加算 (10点/3点/5点)
 - ・合計得点に応じたRESULTパターン (13-28) を選択
- ⑤ GPIO出力
 - ・5bitのpattern_selを各GPIOピンに出力
 - ・例：pattern_sel=13 → BCM17-21に 01101 を出力

10-11. 的当てゲーム：開発の工夫と成果

【技術的チャレンジ】

- ① 大規模LED制御
 - ・9,216ピクセルをリアルタイム制御
 - ・60Hzリフレッシュ
- ② 状態遷移設計
 - ・6状態のFSM
 - ・タイミング厳密制御
- ③ パターン生成
 - ・27種類の画面
 - ・フォント内蔵

【ハイブリッド構成】

- ④ FPGA + ラズパイ連携
 - ・GPIO 5bit通信
 - ・リアルタイム同期
- ⑤ 画像処理連携
 - ・OpenCV処理結果をファイル経由で共有
- ⑥ モジュール設計
 - ・各機能を独立モジュール化
 - ・保守性・拡張性向上

【成果】

- ✓ 技能祭で展示し、来場者に好評
- ✓ FPGAによる大規模LED制御の実践的な経験を獲得
- ✓ ハードウェアとソフトウェアの協調設計スキルを習得
- ✓ Claude AIを活用した効率的な開発プロセスを確立

11-1. 開発プロセスと成果

【段階的开发による品質確保】

Phase 1 (24時間) → Phase 2 (年月日曜日) → Phase 3 (統合)

各フェーズで完全にテスト・検証してから次に進む

C言語による大規模リファレンスデータ生成 + Verilog全網羅テスト

合計 123,290パターン (86,400 + 36,890) の検証に成功

【技術的達成】

16モジュール・850行のVerilogコードを完全自力実装

carry連鎖、ブロックRAM、ダイナミック点灯最適化、MSE統合設計など

複数の高度な技術を駆使

テストベンチのタイミング制御による厳密な検証

(ENABLE negedge → リファレンス読込 → ENABLE posedge → clk negedge → 検証
)

C言語とVerilogの連携による効率的な検証フロー構築

境界値テスト6パターンで特殊ケースも完全検証

11-2. 習得した技術スキル

【ハードウェア設計】

Verilog HDL
階層設計・モジュール分割
状態遷移設計
タイミング制約
ブロックRAM活用
Critical Path最適化
リソース削減技法

【検証・デバッグ】

テストベンチ設計
シミュレーション
全網羅テスト手法
境界値テスト
タイミング解析
論理合成結果の読解
デバッグ技法

【ソフトウェア連携】

C言語プログラミング
リファレンス生成
ファイルI/O
データフォーマット変換
Excelでのデータ分析

【開発プロセス】

段階的開発手法
要件定義
設計・実装・検証サイクル
ドキュメント作成
AI活用による効率化

11-3. 今後の展望

【短期目標（今後3ヶ月）】

残り3機能（アラーム・ストップウォッチ・タイマー）の実装完了
音声出力機能の追加（圧電ブザー、PWM制御でアラーム音・操作音）
電源ON時の初期時刻設定機能の改善（RTCモジュール連携）

【中期目標（今後半年～1年）】

GPS・電波時計モジュールとの連携による自動時刻修正
温度・湿度センサー追加で環境情報表示機能
PCB設計・製作でオリジナル基板を作成し、実用的な時計として完成
組込みLinux（ラズパイ）とFPGAの連携による高度なIoT機器の開発

【キャリア目標】

組込みエンジニアとして、ハードウェアとソフトウェアの両面から製品開発に貢献

11-4. まとめ

【プロジェクトの成果】

- ・101年間動作するデジタル時計をFPGAで完全自力実装
- ・16モジュール、約850行のVerilogコード
- ・123,290パターンの全網羅テストに成功
- ・的当てゲームでFPGA + ラズパイのハイブリッド開発

【習得した技術】

- ・階層設計、モジュール分割、carry連鎖カウンタ設計
- ・ブロックRAM活用、ダイナミック点灯最適化
- ・メタステビリティ・チャタリング・エッジ検出の統合実装
- ・C言語とVerilogの連携による大規模検証フロー構築
- ・テストベンチのタイミング制御による厳密な検証
- ・境界値テストによる特殊ケースの完全検証

【アピールポイント】

- ・完全自力実装：既存コードに頼らず、すべて自分で設計・実装
- ・徹底した品質管理：全パターンテスト + 境界値テスト
- ・問題解決能力：Critical Path短縮、リソース削減など複数の最適化を実施
- ・ドキュメント能力：詳細な技術資料作成、プレゼンテーション能力