前期実験-A3 課題

高山乃綾

2025 年 7 月 18 日

1　実験の概要

本実験では、ハードウェア記述言語 (HDL) を用いて様々な回路を設計し、それに合わせたテストベンチを作成しつつ、波形ビューア GTKWave (gtkwave) などを用いて回路の挙動を視覚的に調べた.

3 日目ではまず、基本的な順序回路である D フリップフロップ (D-FF) と D ラッチ (D-LATCH) の回路を作成し、テストベンチを通して二つの回路素子の振る舞いの違いについて調べた. 次に、クロックに同期した状態リセット信号 (STRAT 信号) と、非同期状態リセット信号 (RST) の回路を設計し、二つの働き方の違いを確認した.

4 日目ではまず、メモリ読み出し回路の設計とシュミレーションを行った. 次に、メモリ読み出し回路に 2 日目に作成したオーバーフロー出力月 4-bit リップルキャリーアダー (4bit-Ripple Carry Adder) を組み合わせ、メモリから読んだ二つのデータを加算していくような回路素子を実現した. 最後に発展課題として、個人的に難しいと感じたブロッキング代入とノンブロッキング代入について、その挙動の違いを示すような回路及びテストベンチを作成した. それにより、HDL の記述の難しさを実感するとともに、テストベンチを作成し波形で動作を確認することの重要性を理解した.

2　3 日目報告

2.1　example モジュールを通した D-FF,D-LATCH の挙動の違い

まず、与えられた example.v のコードは以下のようになる.

各行の説明は、結果で取り上げたい部分以外はコード内のコメントにて記載した.

Code 1　example.v の回路

```
1  `timescale 1ns / 1ps // シュミレーション単位を 1ns 精度を, 1ps に設定
2  module example (
3      input wire inA, // 入力信号
4      input wire clk, // クロック信号
5      output wire out1, // 出力信号 1
6      output wire out2 // 出力信号 2
7  );
8      // 内部レジスタ宣言
9      reg out1_reg, out2_reg;
10
11     always @(posedge clk) begin
```

```
12          // out1_reg - クロックの立ち上がりに応じて inA をキャプチャする
13          out1_reg <= inA;
14      end
15
16      // クロックが High のときだけ inA を out2_reg に代入
17      always @(*) begin
18          if (clk==1'b1)
19             out2_reg <= inA;
20      end
21
22      assign out1 = out1_reg; // 内部レジスタ out1_reg の値を出力ポート out1 に接続
23      assign out2 = out2_reg; // 内部レジスタ out2_reg の値を出力ポート out2 に接続
24
25  endmodule
```

これに対し、自作のテストベンチファイル tb_example.v を以下のように作成した.

Code 2　自作テストベンチ tb_example.v

```
1   /*シュミレーション単位: 1ns 精度,: 1ps に設定*/
2
3   `timescale 1ns / 1ps
4
5   module tb_example;
6
7       // モニタリング用に波形を VCD 出力
8       initial begin
9           $dumpfile("tb_example.vcd"); // 出力ファイル名
10          $dumpvars(0, tb_example);
11      end
12
13      reg inA; // DUT の inA に接続
14      reg clk; // DUT の clk に接続
15
16      wire out1; // DUT の out1 を受けとる
17      wire out2; // DUT の out2 を受け取る
18
19      // DUT のインスタンス化
20      // モジュールを example uut (unit under test) として呼び出す
21      example uut (
22          .inA(inA),
23          .clk(clk),
24          .out1(out1),
25          .out2(out2)
26      );
27
28      // クロック生成: 周期 10ns でクロックを発生
29      initial begin
30          clk = 0; // 初期値 Low に設定
31          forever #5 clk = ~clk;
32          // 5ns ごとに反転 → 周期 10ns のクロック
33      end
34
35      // テストパターンの生成
36      /* inA に対して異なるタイミングで High/Low を設定することで、
```

```
37      out1 と out2 の違いを見る */
38
39      initial begin
40          inA = 0;
41          #12  inA = 1; // 12ns で Low → High
42          #18  inA = 0; // 30ns で High → Low
43          #22  inA = 1; // 52ns で Low → High
44          #16  inA = 0; // 68ns で High → Low
45          #20  inA = 1; // 88ns で Low → High
46          #50  $finish; // 合計 138ns でシュミレーション終了
47      end
48
49      // モニタリング: シュミレーション時間と信号を表示
50      initial begin
51          $display("time\tclk inA out1 out2");
52          $monitor("%0dns\t%b   %b    %b    %b",
53                   $time, clk, inA, out1, out2);
54      end
55
56  endmodule
```

example.v と tb_example.v を用いて gtkwave 上で波形を表示させたところ、 clk, inA, OUT1, OUT2 はそれぞれ下の図のように動いた.

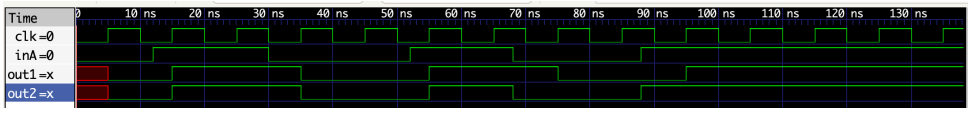

図 1　OUT1 と OUT2 の波形の違い

図１から読み取れること

OUT1 と OUT2 の違いは、特に $t = 68$ ns と $t = 88$ ns で顕著に見られる.

- $t = 68$ ns **の場合**: OUT2 はこの時点で inA の変化に瞬時に反応しているが、OUT1 は次の立ち上がりエッジ $t = 75$ ns まで前の値を保持している.
- $t = 88$ ns **の場合**: このとき clk $= 1$ であり、inA が立ち上がると OUT2 は即座に inA に追従して立ち上がる. 一方 OUT1 は、clk の立ち上がりしか見ないため、次の立ち上がりエッジ $t = 95$ ns まで値が 0 のままである.

以上により、レベルが High の間ずっと入力を通す OUT2 が D-LATCH であり、立ち上がりエッジにしか感応しない OUT1 が D-FF だと判断できる.

2.2　同期・非同期リセット付きシーケンスジェネレータの設計

まず、状態遷移回路本体（seq_gen.v）のコードは以下のとおり. 主要部分にはコメントを入れている.

Code 3　シーケンスジェネレータ本体 seq_gen.v

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

```
2
3  module seq_gen (
4      input  wire       clk,    // クロック
5      input  wire       rst,    // 非同期リセット (1 で即座に S1 へ)
6      input  wire       START,  // 同期リセット (posedge clk で S1 へ)
7      output reg        out     // S1 のときだけ 1 を出力
8  );
9
10     // 状態定義
11     localparam S1 = 2'd0,
12                S2 = 2'd1,
13                S3 = 2'd2,
14                S4 = 2'd3;
15
16     reg [1:0] state, next_state;
17
18     // 次状態ロジック 非同期リセット優先 ( → 同期リセット → 通常遷移)
19     always @(*) begin
20         if (START) begin
21             next_state = S1;
22         end else begin
23             case (state)
24                 S1: next_state = S2;
25                 S2: next_state = S3;
26                 S3: next_state = S4;
27                 S4: next_state = S1;
28                 default: next_state = S1;
29             endcase
30         end
31     end
32
33     // 状態レジスタ＆非同期リセット
34     always @(posedge clk or posedge rst) begin
35         if (rst) begin
36             state <= S1;           // rst=1 で即座に S1 へ
37         end else begin
38             state <= next_state;   // clk エッジで遷移
39         end
40     end
41
42     // 出力ロジック 状態が ( S1 のときだけ 1)
43     always @(state) begin
44         out = (state == S1) ? 1'b1 : 1'b0;
45     end
46
47 endmodule
```

つぎにテストベンチ（tb_seq_gen.v)。同期リセットと非同期リセットの挙動を観測するため、波形ダンプと$monitor を設定している.

Code 4　テストベンチ tb_seq_gen.v

```
1  `timescale 1ns / 1ps
2
3  module tb_seq_gen;
```

```
    reg  clk;
    reg  rst;
    reg  START;
    wire out;

    // DUT インスタンス化
    seq_gen uut (
        .clk   (clk),
        .rst   (rst),
        .START (START),
        .out   (out)
    );

    // 波形ダンプ & モニタ
    initial begin
        $dumpfile("tb_seq_gen.vcd");
        $dumpvars(0, tb_seq_gen);
        $display("time\tclk rst START out");
        $monitor("%0t\t%b   %b     %b    %b",
                 $time, clk, rst, START, out);
    end

    // クロック生成 (10ns 周期)
    initial begin
        clk = 1'b0;
        forever #5 clk = ~clk;
    end

    // 非同期リセット動作確認
    initial begin
        rst   = 1'b1;   // アサート
        START = 1'b0;
        #8    rst = 1'b0; // 解除 → state=S1, out=1
    end

    // 同期リセットと非同期リセットのテストシーケンス
    initial begin
        #20;
        // -- 同期リセット (START) --
        @(posedge clk);
        START = 1'b1;
        @(posedge clk);
        START = 1'b0;
        repeat (4) @(posedge clk);

        #20;
        // -- 非同期リセット (rst) --
        rst = 1'b1;     // 即座にリセット
        #3  rst = 1'b0;
        @(posedge clk);
        repeat (2) @(posedge clk);

        #20;
        $finish;
    end
endmodule
```

これを iverilog → vvp → gtkwave でシミュレーションし、得られた波形を図 2 に示す.

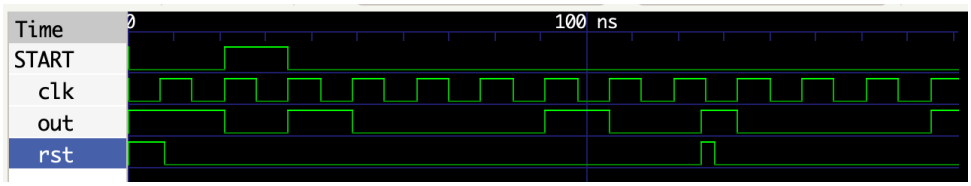

図 2　同期リセット (START) と非同期リセット (rst) の動作比較

図 2 から読み取れること

- 同期リセット (START)：START=1 を与えたクロック立ち上がり時にのみ状態 S1 へ戻る. そのあとは通常遷移（S1 → S2 →…）をクロック同期でくり返す.
- 非同期リセット (rst)：rst=1 になるとクロックに関係なく即座に状態 S1 へ遷移。解除後、最初の立ち上がりエッジで再び通常遷移が始まる.

3　4 日目報告

3.1　メモリ読み出し + 加算回路の設計とシミュレーション

3.1.1　BRAM 例の回路 (bram_example.v)

BRAM にあらかじめロードした mem.hex の内容をアドレス 0 から順に読み出す回路を設計した. メモリは Read-First モードで、クロック立ち上がりで読み書き同期を取る.

Code 5　BRAM 例 bram_example.v

```
1  `timescale 1ns / 1ps
2  module bram_example (
3      input  wire       clk,    // クロックで読み書き同期
4      input  wire       we,     // 書き込み許可
5      input  wire [4:0] addr,   // 5bit アドレス (0--31)
6      input  wire [3:0] w_data, // 4bit 書込データ
7      output wire [3:0] r_data  // 4bit 読出データ
8  );
9      reg [3:0] mem [0:31];
10     reg [3:0] mem_out;
11
12     initial begin
13         $readmemh("mem.hex", mem);
14     end
15
16     always @(posedge clk) begin
17         if (we)
18             mem[addr] <= w_data;
19         mem_out <= mem[addr];
20     end
21
22     assign r_data = mem_out;
23 endmodule
```

mem.hex のサンプル（先頭 8 文字のみ）{6, C, F, A, B, C, 3, 9, ...} ... と 256 個の数字が縦に並んでいる.

3.1.2 読み出し＋加算回路とテストベンチ (mem_reader.v, tb_mem_reader.v)

読み出したデータを加算し、最後に done をアサートする回路を作成. テストベンチでアドレス順に値を表示し、波形と動作を確認した.

Code 6 メモリ読み出し＋カウント回路 mem_reader.v

```
module mem_reader (
    input  wire       clk,
    input  wire       rst_n,   // 非同期リセット（リセット）0:
    output wire [3:0] data_o,  // 現在読み出したデータ
    output wire       done     // 全語読み終わったら 1
);
    //---------------------------------------------
    // アドレスカウンタ (0^^e2^^80^^9331)
    //---------------------------------------------
    reg  [5:0] addr_cnt;       // 6bit にしておくと カウント後に 32 32 になる
    wire reach_end = (addr_cnt == 6'd32); // 下位 5だけをアドレスに用いる bit

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            addr_cnt <= 0;      // rst_n=0 で即座にクリア
        else if (!reach_end)
            addr_cnt <= addr_cnt + 1; // 32 未満ならインクリメント
    end

    //---------------------------------------------
    // BRAM 読み出しインスタンス
    //---------------------------------------------
    bram_example bram_i (
        .clk    (clk),
        .we     (1'b0),           // 書き込み抑止
        .addr   (addr_cnt[4:0]),  // アドレスは下位 5bit
        .w_data (4'b0),
        .r_data (data_o)
    );

    //---------------------------------------------
    // 終了フラグ
    //---------------------------------------------
    assign done = reach_end;
endmodule
```

Code 7 テストベンチ tb_mem_reader.v

```
`timescale 1ns / 1ps
module tb_mem_reader;
```

```
    // クロック 100MHz (10ns)
    reg clk = 0;        always #5 clk = ~clk;

    // DUT 接続
    reg        rst_n = 0;
    wire [3:0] data;
    wire       done;
    mem_reader dut (
        .clk    (clk),
        .rst_n  (rst_n),
        .data_o (data),
        .done   (done)
    );

    // 波形ダンプ
    initial begin
        $dumpfile("sim1.vcd");
        $dumpvars(0, tb_mem_reader);
    end

    // テストシーケンス
    integer i;
    reg first = 1;
    reg [4:0] addr_prev;
    initial begin
        #12 rst_n = 1;          // リセット解除
        $display("addr : data");
        for (i = 0; i < 32; i = i + 1) begin
            @(posedge clk);
            if (!first)
                $display("%0d : %h", addr_prev, data);
            addr_prev <= dut.addr_cnt;
            first <= 0;
        end
        @(posedge clk);
        $display("%0d : %h", addr_prev, data);
        if (done) $display("=== DONE ===");
        $finish;
    end
endmodule
```

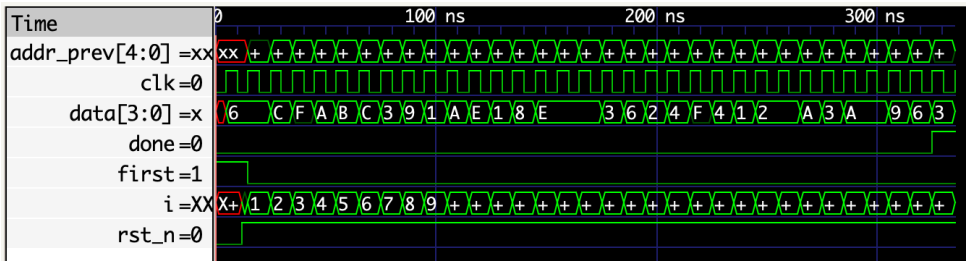

図 3　メモリ読み出し＋加算回路の波形

図 3 から読み取れること

- アドレス 0 → 31 のレンジで、各クロック立ち上がり後に data_o が正しく出力されている.
- 1 クロック遅れで出力するため、表示が常にひとつ前のアドレスに対応している.
- 最後に done がアサートされ、ループ終了が検知できている.

3.2 メモリ読み出し＋ 4-bit 加算回路

3.2.1 狙いと構成

アドレス n と $n+1$ の 2 語を同時に読み出し，4-bit リップルキャリーアダー (rca4.v) で加算して和 (sum) とオーバーフロー (ov) を得る．これを 16 ペア（0–1, 1–2, ⋯ , 15–16）について行い，参照 BRAM (bram_ref.v) と照合して動作を確かめる.

3.2.2 4-bit Ripple-Carry Adder (rca4.v)

rca4.v は 2 日目に作成した半加算器→全加算器→ 4 段直列の標準的な RCA.MSB へのキャリーと MSB からのキャリーの XOR で符号付きオーバーフローを検出する.

3.2.3 連続 2 語読み出し回路 (mem_pair_reader.v)

まずペア読み出し専用回路を実装した（Listing 8).

Code 8　ペア読み出し回路 mem_pair_reader.v

```
1  `timescale 1ns / 1ps
2  module mem_pair_reader(
3      input  wire       clk,
4      input  wire       rst_n,
5      output reg  [3:0] a, b,
6      output reg  [4:0] pair_addr_d,
7      output wire       done
8  );
9      // ペアアドレス →→⋯→→ 011516
10     reg [4:0] pair_addr;
11     always @(posedge clk or negedge rst_n) begin
12         if (!rst_n) begin
13             pair_addr   <= 5'd0;
14             pair_addr_d <= 5'd0;
15         end else begin
16             pair_addr_d <= pair_addr;          // 1 clk 遅延で外部へ
17             pair_addr   <= pair_addr + 5'd1;  // 1 語ずつスライド
18         end
19     end
20     assign done = (pair_addr == 5'd16);
21
22     // BRAM から 2 語同時に読み出し
23     wire [3:0] d0, d1;
24     bram_example mem0 (.clk(clk), .we(1'b0),
25                        .addr(pair_addr      ), .w_data(4'b0), .r_data(d0));
26     bram_example mem1 (.clk(clk), .we(1'b0),
27                        .addr(pair_addr + 5'd1), .w_data(4'b0), .r_data(d1));
28
29     always @(posedge clk) begin
```

```
30          a <= d0;
31          b <= d1;
32      end
33  endmodule
```

■要点

- 1 クロックごとに $\{n, n+1\}$ の 2 語を読み出し，1 クロック遅延させて a,b と pair_addr_d を同期.
- 16 ペア処理し終えると done=1 を返して処理完了.

3.2.4 検証ベンチ (tb_add_check.v)

DUT (mem_pair_reader + rca4) と参照 BRAM (bram_ref.v) を比較し，差異があれば fail=1 とする.

3.2.5 シミュレーション結果

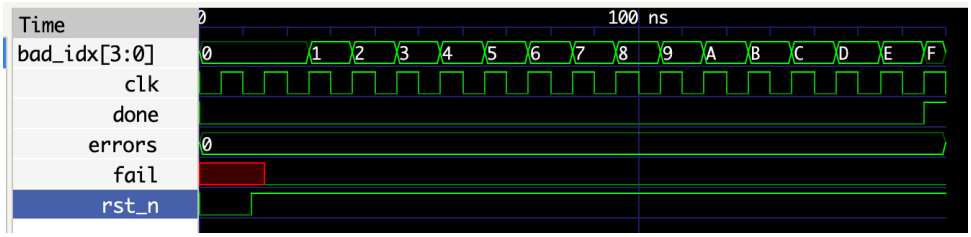

図 4　ペア読み出し＋加算結果と参照比較の波形

図 4 から分かること

- pair_addr_d=0–15 が現れるたび，同期して a,b → sum,ov が更新されている.
- 参照比較フラグ fail は常に Low。DUT と参照出力が完全一致.
- 16 ペア目が終わると done=1 となり，シミュレーション終了.

3.3 発展課題——ブロッキング代入とノンブロッキング代入

3.3.1 概　要

Verilog には

- **ブロッキング代入**　（=）右辺を評価し直ちに左辺へ格納した後，次の文へ進む
- **ノンブロッキング代入**（<=）右辺を評価した値を時刻ステップの終端で一括してレジスタへ反映

という 2 種類の代入方式が存在する．組み合わせ回路にはブロッキング，順序（レジスタ）回路にはノンブロッキングを用いるのが原則であるが，規則を誤った場合，シミュレーション結果と本来意図したタイミングが一致しないことがある．以下に最小回路を示し，両方式の違いが波形上でどのように現れるかを検証した.

3.3.2 比較用ミニ回路 nb_vs_b.v

Code 9　ブロッキング代入とノンブロッキング代入の比較用回路

```
`timescale 1ns / 1ps
module nb_vs_b (
    input  wire clk,   // 10-ns 周期
    input  wire d,
    output reg  q2_nb, // 正しい遅延 () <=
    output reg  q2_b   // 誤った遅延 () =
);
    // ---- ノンブロッキング版（正） ----
    reg q1_nb;
    always @(posedge clk) begin
        q1_nb <= d;        // 1 段目
        q2_nb <= q1_nb;    // 2 段目（1 clk 遅延）
    end

    // ---- ブロッキング版（誤） ----
    reg q1_b;
    always @(posedge clk) begin
        q1_b = d;          // 直ちに上書き
        q2_b = q1_b;       // 同じクロック内で伝搬
    end
endmodule
```

■動作の意図　信号 d を 2 クロック遅らせて q2_nb ／ q2_b に出力したい．本来はノンブロッキング（<=）で 2 段遅延を実現すべきところを，意図的にブロッキング（=）へ置き換え，遅延動作の崩れを観察する．

3.3.3　テストベンチ tb_nb_vs_b.v

Code 10　比較用テストベンチ

```
`timescale 1ns / 1ps
module tb_nb_vs_b;
    // 10-ns クロック生成
    reg clk = 0; always #5 clk = ~clk;

    // 入力パルス
    reg d = 0;
    initial begin
        #12 d = 1;          // 幅 10 ns
        #10 d = 0;
        #100 $finish;
    end

    // デバイス・アンダ・テスト
    wire q2_nb, q2_b;
    nb_vs_b dut (.clk(clk), .d(d), .q2_nb(q2_nb), .q2_b(q2_b));

    // 波形ダンプ
    initial begin
        $dumpfile("nb_vs_b.vcd");
        $dumpvars(0, tb_nb_vs_b);
    end
endmodule
```

3.3.4　シミュレーション結果

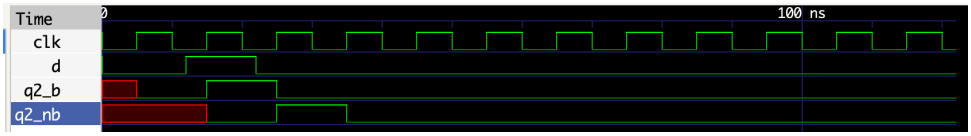

図 5　ブロッキング代入とノンブロッキング代入の遅延差

■図 5 の考察

- 入力 d のパルスに対し，**ノンブロッキング版** q2_nb は 2 クロック（20 ns）遅延して出力されており，設計意図どおりの動作が得られている．
- **ブロッキング版** q2_b は同一クロック内で q1_b が直ちに書き換わるため，遅延なしに d をコピーしてしまう．
- 逐次回路をブロッキング代入で記述すると，シミュレーション上は動作するように見えても，実際の回路タイミングは崩れ，意図しない競合やレースの原因となる．

3.3.5　まとめ

- クロック同期レジスタは **ノンブロッキング代入（<=|）** を用いることで，同じ時刻内での値の書き換え競合を防止できる．
- 組み合わせロジックやテストベンチの一時変数は**ブロッキング代入（=|）** を用いると可読性が高い．
- 本検証により，代入方式の誤用が波形にどのような影響を及ぼすかを可視化できた。設計段階で代入方式を明確に区別することの重要性が確認された．

全課題のまとめ

今回の実験では，Verilog の基礎から少し発展的な内容まで段階的に確認した．ポイントは次の 4 つである．

D-FF と D-Latch の違いエッジ感度（D-FF）とレベル感度（D-Latch）の動きを同じ波形上で比べ，クロックの立ち上がりだけ反応するか，High の間ずっと追従するかを目で確かめた．

同期リセットと非同期リセット同期リセットはクロックに合わせて状態が戻り，非同期リセットはクロックを待たずに即リセットされることをシミュレーションで確認した．

メモリ読み出しと加算回路 BRAM からデータを読み出し，4-bit リップルキャリーアダーで順に足し合わせる回路を作成. 参照メモリと比較し，すべての加算結果が一致することを波形で確認した．

ブロッキング代入とノンブロッキング代入同じレジスタ遅延をブロッキング（=）で書くと遅延がなくなり，ノンブロッキング（¡=）で書くと正しく遅延することを最小回路で検証した。クロック同期の always では <= を使う必要があることがはっきり分かった．

以上の結果から，設計した HDL が意図どおり動作しているかを波形で確認することの大切さと，代入記号など基本ルールを守る重要性を実感した．