Table des matières

INTRODUCTION

Ce rapport présente le travail réalisé dans le cadre de notre projet d'Architecture des ordinateurs, une initiation à la conception numérique, avec pour objectif la compréhension, l'implémentation et la simulation de composants logiques fondamentaux. Ces derniers constituent les briques de base de tout système numérique, qu'il s'agisse de microprocesseurs, de contrôleurs logiques ou de systèmes embarqués.

Trois composants essentiels ont été étudiés et implémentés à l'aide du langage **Verilog HDL** :

- L'**additionneur binaire** : un circuit combinatoire permettant d'effectuer des opérations d'addition sur des mots binaires.
- Le **décodeur 2 vers 4** : un circuit qui convertit un code binaire de 2 bits en une sortie unique parmi 4 lignes.
- Le **registre à décalage** (vers la droite et vers la gauche) : un circuit séquentiel permettant de décaler des données binaires dans un registre à chaque front d'horloge.

Ces composants sont couramment utilisés dans les systèmes embarqués, les unités de calcul (ALU), ainsi que dans les circuits de contrôle et d'interface.

Pour chacun de ces éléments, ce rapport propose :

- Une **présentation théorique** : principe de fonctionnement, utilité, symboles et table de vérité.
- Une **illustration** graphique ou un exemple de fonctionnement.
- Une **implémentation en Verilog HDL**, respectant une structure claire et modulaire.
- Un **banc d'essai** (*testbench*) permettant de simuler et valider le comportement du circuit.

Le code source complet, accompagné des fichiers de simulation et des diagrammes d'ondes compatibles avec *GTKWave*, est disponible sur le dépôt GitHub suivant :

`https://www.github.com/Maruba22/computerdesign2025`

Les simulations ont été réalisées à l'aide des outils **Icarus Verilog** et **GTKWave**, qui permettent respectivement la compilation du code et l'analyse temporelle des signaux logiques.

Ce projet nous a permis de renforcer notre compréhension des circuits logiques, de développer des compétences pratiques en programmation Verilog, et d'approfondir notre maîtrise du fonctionnement interne des architectures numériques modernes.

Question 1 : Additionneur Binaire

i) Qu'est-ce qu'un additionneur binaire et quelle est son utilité dans les circuits numériques ?

Un **additionneur binaire** est un circuit logique combinatoire qui effectue l'addition de deux nombres binaires. Il est largement utilisé dans les unités arithmétiques et logiques (ALU) des processeurs, ainsi que dans divers systèmes embarqués nécessitant des calculs binaires.

On distingue principalement deux types d'additionneurs :

- **Demi-additionneur (Half Adder)** : il réalise l'addition de deux bits d'opérande sans tenir compte d'une éventuelle retenue en entrée. Il produit deux sorties : la somme (`Sum`) et la retenue (`Carry`).
- **Additionneur complet (Full Adder)** : il effectue l'addition de deux bits et d'une retenue d'entrée. Il fournit une somme (`Sum`) et une retenue de sortie (`Carry Out`).

Half Adder

a	*b*	*sum*	*carry-out*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

a	*b*	*cin*	*sum*	*carry-out*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 1 : Représentation du demi-additionneur et de l'additionneur complet

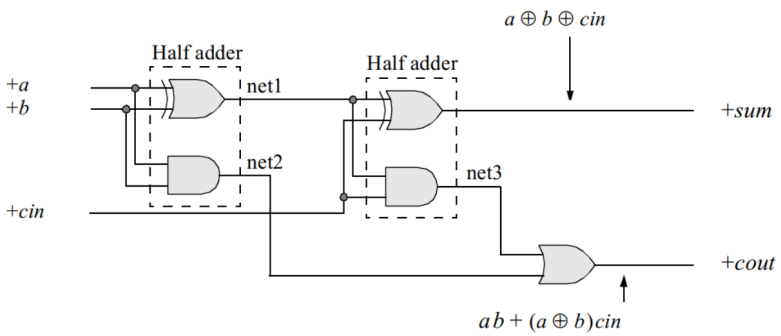

Figure 2 : Schéma logique d'un additionneur complet

ii) Comment concevoir un additionneur binaire de n bits ? Quels sont les principaux signaux d'entrée et de sortie ?

Un **additionneur binaire de n bits** est constitué d'une cascade de n additionneurs complets, chacun traitant un bit de deux opérandes binaires A et B, ainsi qu'une retenue (ou *carry*) issue de l'étape précédente.

Les **signaux principaux** sont :

- **Entrées** :
  - `A[N-1:0]` : premier opérande binaire de n bits,
  - `B[N-1:0]` : second opérande binaire de n bits,
  - `Cin` : retenue d'entrée (carry-in), utilisée pour l'addition en cascade.
- **Sorties** :
  - `S[N-1:0]` : résultat de la somme binaire,
  - `Cout` : retenue de sortie (carry-out), qui indique un dépassement de capacité.

iii) Quelle est la différence entre un additionneur à retenue et un additionneur sans retenue ?

La différence principale réside dans la gestion de la **retenue** (*carry*) entre les bits :

- Un **additionneur à retenue** (*carry-aware*) tient compte des retenues générées entre les bits. Il est donc capable d'effectuer correctement l'addition de nombres binaires multi-bits. C'est le cas de l'additionneur en cascade (*Ripple Carry Adder*), où chaque retenue est transmise à l'étape suivante.
- Un **additionneur sans retenue** n'intègre pas la propagation des retenues entre bits. Chaque bit est traité indépendamment, ce qui limite son usage à des opérations très simples (souvent sur un seul bit).

MODULES , TESTBENCH ET SIMULATION

1. Demi-Additionneur

```
module half_adder (
    input A,
    input B,
    output Sum,
    output Cout
);
    assign Sum = A ^ B;     // somme = A XOR B
    assign Cout = A & B;    // retenue = A AND B
endmodule
```

```
module half_adder_tb;
    reg A, B;
    wire Sum, Cout;

    half_adder uut(.A(A), .B(B), .Sum(Sum), .Cout(Cout));

    initial begin
        $display("A B | Sum Cout");
        A=0; B=0; #5 $display("%b %b |  %b    %b", A, B, Sum, Cout);
        A=0; B=1; #5 $display("%b %b |  %b    %b", A, B, Sum, Cout);
        A=1; B=0; #5 $display("%b %b |  %b    %b", A, B, Sum, Cout);
        A=1; B=1; #5 $display("%b %b |  %b    %b", A, B, Sum, Cout);
        $finish;
    end
endmodule
```

```
PS E:\maruba\computerdesign2025> cd .\adder\
PS E:\maruba\computerdesign2025\adder> iverilog -o sim.out half_adder.v testbench/half_adder_tb.v
PS E:\maruba\computerdesign2025\adder> vvp sim.out
 A B | Sum Cout
 0 0 |  0   0
 0 1 |  1   0
 1 0 |  1   0
 1 1 |  0   1
 testbench/half_adder_tb.v:13: $finish called at 20 (1s)
```

2. Additionneur Complet

```
module full_adder (
    input A,
    input B,
    input Cin,
    output Sum,
    output Cout
);
    assign Sum = A ^ B ^ Cin;
    assign Cout = (A & B) | (B & Cin) | (A & Cin);
endmodule
```

```
module full_adder_tb;
    reg A, B, Cin;
    wire Sum, Cout;

    full_adder uut(.A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(Cout));

    initial begin
        $display("A B Cin | Sum Cout");
        for (integer i=0; i<8; i=i+1) begin
            {A,B,Cin} = i;
            #5;
            $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);
        end
        $finish;
    end
endmodule
```

```
PS E:\maruba\computerdesign2025\adder> iverilog -o sim.out full_adder.v testbench/full_adder_tb.v
PS E:\maruba\computerdesign2025\adder> vvp sim.out
A B Cin | Sum Cout
0 0  0  |  0   0
0 0  1  |  1   0
0 1  0  |  1   0
0 1  1  |  0   1
1 0  0  |  1   0
1 0  1  |  0   1
1 1  0  |  0   1
1 1  1  |  1   1
testbench/full_adder_tb.v:14: $finish called at 40 (1s)
```

3. Additionneur avec anticipation

```
module carry_lookahead_adder #(parameter N=4) (
    input  [N-1:0] A,
    input  [N-1:0] B,
    input          Cin,
    output [N-1:0] Sum,
    output         Cout
);
    wire [N-1:0] G, P;
    wire [N:0] C;
```

```
    assign G = A & B;
    assign P = A ^ B;
    assign C[0] = Cin;

    genvar i;
    generate
        for(i=0; i<N; i=i+1) begin : carry_calc
            assign C[i+1] = G[i] | (P[i] & C[i]);
        end
    endgenerate

    assign Sum = P ^ C[N-1:0];
    assign Cout = C[N];
endmodule
```

```
module carry_lookahead_adder_tb;
    parameter N=4;
    reg  [N-1:0] A, B;
    reg          Cin;
    wire [N-1:0] Sum;
    wire         Cout;

    carry_lookahead_adder #(N) uut(.A(A), .B(B), .Cin(Cin), .Sum(Sum), .
       Cout(Cout));

    initial begin
        $display("A    B    Cin | Sum  Cout");
        A = 4'b0000; B = 4'b0000; Cin = 0; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        A = 4'b0110; B = 4'b0011; Cin = 1; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        A = 4'b1111; B = 4'b1111; Cin = 0; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        A = 4'b1010; B = 4'b0101; Cin = 1; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        $finish;
    end
endmodule
```

```
PS E:\maruba\computerdesign2025\adder> iverilog -o sim.out carry_look_ahead.v testbench/carry_look_ahead_tb.v
PS E:\maruba\computerdesign2025\adder> vvp sim.out
A    B    Cin | Sum  Cout
0000 0000  0  | 0000  0
0110 0011  1  | 1010  0
1111 1111  0  | 1110  1
1010 0101  1  | 0000  1
testbench/carry_look_ahead_tb.v:16: $finish called at 40 (1s)
PS E:\maruba\computerdesign2025\adder>
```

4. Additionneur à propagation de retenue

```
module ripple_carry_adder #(parameter N = 4) (
    input  [N-1:0] A,
    input  [N-1:0] B,
    input          Cin,
    output [N-1:0] Sum,
    output         Cout
);
    wire [N:0] carry;
    assign carry[0] = Cin;

    genvar i;
    generate
        for (i=0; i < N; i=i+1) begin : full_adders
            full_adder fa (
                .A(A[i]),
                .B(B[i]),
                .Cin(carry[i]),
                .Sum(Sum[i]),
                .Cout(carry[i+1])
            );
        end
    endgenerate

    assign Cout = carry[N];
endmodule
```

```
module ripple_carry_adder_tb;
    parameter N=4;
    reg  [N-1:0] A, B;
    reg          Cin;
    wire [N-1:0] Sum;
    wire         Cout;

    ripple_carry_adder #(N) uut(.A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(
       Cout));

    initial begin
        $display("A    B    Cin | Sum  Cout");
        A = 4'b0000; B = 4'b0000; Cin = 0; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        A = 4'b0011; B = 4'b0101; Cin = 0; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        A = 4'b1111; B = 4'b0001; Cin = 0; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        A = 4'b1010; B = 4'b0101; Cin = 1; #10 $display("%b %b  %b  | %b
             %b", A, B, Cin, Sum, Cout);
        $finish;
    end
endmodule
```

```
PS E:\maruba\computerdesign2025\adder> iverilog -o sim.out ripple_carry_adder.v full_adder.v testbench/ripple_carry_adder_tb.v
PS E:\maruba\computerdesign2025\adder> vvp sim.out
  A    B    Cin | Sum  Cout
  0000 0000  0  | 0000   0
  0011 0101  0  | 1000   0
  1111 0001  0  | 0000   1
  1010 0101  1  | 0000   1
  testbench/ripple_carry_adder_tb.v:16: $finish called at 40 (1s)
PS E:\maruba\computerdesign2025\adder>
```

5. Additionneur en pipeline

```
module pipelined_adder #(parameter N=8, STAGES=2) (
    input clk,
    input reset,
    input [N-1:0] A,
    input [N-1:0] B,
    input Cin,
    output reg [N-1:0] Sum,
    output reg Cout
);
    reg [N/2-1:0] sum_low;
    reg carry_mid;
    reg [N/2-1:0] sum_high;
    reg carry_out;

    always @(posedge clk or posedge reset) begin
        if(reset) begin
            sum_low <= 0;
            carry_mid <= 0;
            sum_high <= 0;
            carry_out <= 0;
            Sum <= 0;
            Cout <= 0;
        end else begin
            {carry_mid, sum_low} <= A[N/2-1:0] + B[N/2-1:0] + Cin;
            {carry_out, sum_high} <= A[N-1:N/2] + B[N-1:N/2] + carry_mid;
            Sum <= {sum_high, sum_low};
            Cout <= carry_out;
        end
    end
endmodule
```

```
`timescale 1ns/1ps
module pipelined_adder_tb;
    parameter N = 8;
    reg clk, reset;
    reg [N-1:0] A, B;
    reg Cin;
    wire [N-1:0] Sum;
    wire Cout;

    pipelined_adder #(N, 2) uut (
        .clk(clk),
        .reset(reset),
        .A(A),
```

```
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    initial clk = 0;
    always #5 clk = ~clk; // 100MHz

    initial begin
        $dumpfile("pipelined_adder.vcd");
        $dumpvars(0, pipelined_adder_tb);
    end

    initial begin
        reset = 1;
        A = 0; B = 0; Cin = 0;
        #10; reset = 0;

        A = 8'd10;  B = 8'd20;  Cin = 0; #10;
        A = 8'd255; B = 8'd1;   Cin = 1; #10;
        A = 8'd128; B = 8'd128; Cin = 0; #10;
        A = 8'd100; B = 8'd55;  Cin = 0; #10;
        A = 8'd127; B = 8'd1;   Cin = 1; #10;

        #30;

        $display("Dernier r sultat : A = %d, B = %d, Cin = %b => Sum = %d
           , Cout = %b",
                 A, B, Cin, Sum, Cout);
        $finish;
    end
endmodule
```

```
PS E:\maruba\computerdesign2025\adder> iverilog -o sim.out pipelined_adder.v  testbench/pipelined_adder_tb.v
PS E:\maruba\computerdesign2025\adder> vvp sim.out
VCD info: dumpfile pipelined_adder.vcd opened for output.
Time=22000 | A=  0 B=  0 Cin=0 -> Sum=  0 Cout=0
Time=32000 | A= 10 B= 20 Cin=0 -> Sum=  0 Cout=0
Time=42000 | A=255 B=  1 Cin=1 -> Sum= 30 Cout=0
Time=52000 | A=128 B=128 Cin=0 -> Sum=241 Cout=0
Time=62000 | A=200 B= 56 Cin=0 -> Sum= 16 Cout=1
Time=72000 | A=127 B=  1 Cin=1 -> Sum=240 Cout=0
Time=82000 | A=255 B=255 Cin=1 -> Sum=129 Cout=0
testbench/pipelined_adder_tb.v:53: $finish called at 122000 (1ps)
PS E:\maruba\computerdesign2025\adder>
```

Question 4 : Décodeur 2 à 4

4.1 Rôle du décodeur

Un **décodeur 2 à 4** est un circuit logique combinatoire qui active une seule ligne de sortie parmi quatre, selon la valeur binaire présente sur ses deux entrées. Chaque combinaison binaire unique active exactement une sortie différente, les autres restant désactivées.

Le décodeur est utilisé notamment pour :
- La sélection d'une ligne mémoire (dans un système d'adressage).
- L'activation d'un périphérique spécifique dans un système embarqué.

Exemple : Si l'entrée vaut `10` (soit 2 en décimal), alors la sortie active sera `0100` (la troisième ligne).

4.2 Utilisation dans les systèmes numériques

Les décodeurs sont omniprésents dans la conception des systèmes numériques pour :
- Le **décodage d'adresses**, notamment dans les systèmes à mémoire (RAM, ROM, Flash).
- La **gestion des ressources partagées**, par exemple dans un système multipériphérique.
- Le **décodage d'instructions** dans les microprocesseurs ou automates à états finis.

4.3 Différences avec un multiplexeur

(a) Multiplexeur (MUX)

Un **multiplexeur** (MUX) est un circuit combinatoire qui permet de diriger une entrée parmi plusieurs vers une unique sortie, selon des signaux de sélection. Il joue le rôle d'un commutateur contrôlé par bits.

Les signaux de sélection sont notés $s_0, s_1, ..., s_{n-1}$ et les entrées de données sont $d_0, d_1, ..., d_{2^n-1}$. Si $n = 2$, alors le multiplexeur aura 4 entrées de données (d_0 à d_3) et 2 lignes de sélection (s_0, s_1).

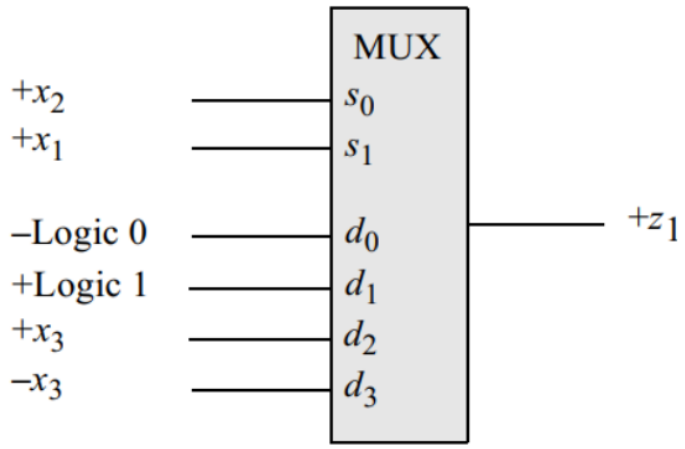

Figure : Schéma d'un multiplexeur 4 :1

Il existe deux types de multiplexeurs :
- **Multiplexeurs à sélection linéaire** : utilisent toutes les variables comme sélecteurs, chaque minterme correspond à une entrée. Ce type est simple mais parfois coûteux en ressources.
- **Multiplexeurs à sélection non linéaire** : utilisent seulement une partie des variables pour simplifier la logique et réduire la complexité matérielle.

(b) Décodeur

Un **décodeur** est un circuit combinatoire qui active une seule sortie pour chaque combinaison binaire unique d'entrées. Contrairement au multiplexeur, il fonctionne comme un décodeur d'état : chaque sortie correspond à un minterme de la table de vérité.

Un décodeur $n : 2^n$ possède n entrées et 2^n sorties, dont une seule est activée à la fois.

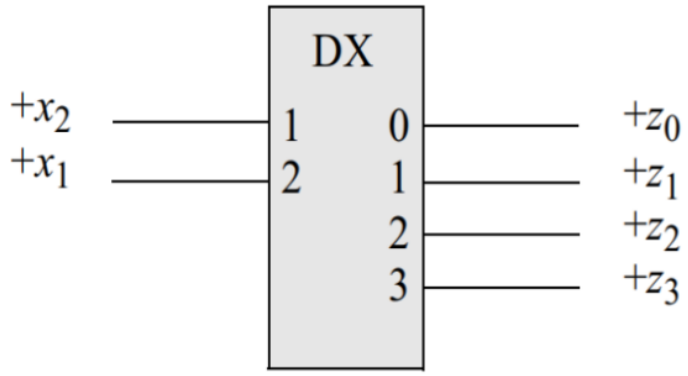

Figure : Schéma d'un décodeur $n : 2^n$

Chaque sortie z_i est active si les entrées représentent le minterme m_i. Ce principe est utilisé dans :

- Le décodage d'instructions (ex. : opcodes de processeurs),
- L'adressage mémoire (activation d'une cellule ou d'un bloc),
- Les automates et systèmes séquentiels (états ou transitions).

Caractéristiques	Décodeur	Multiplexeur
Fonction principale	Active une seule sortie selon l'entrée binaire	Sélectionne une entrée de données vers une sortie unique
Structure	n entrées binaires $\Rightarrow$ 2^n sorties	2^n entrées de données, n sélecteurs, 1 sortie
Nombre de sorties	Plusieurs sorties (une seule active)	Une seule sortie
Utilisation typique	Décodage d'adresses, activation de lignes mémoire ou périphériques	Acheminement d'une des entrées vers la sortie (sélection de données)
Direction du flux	Entrées $\rightarrow$ sorties (activation)	Une des entrées $\rightarrow$ sortie (données)
Exemple de fonctionnement	Entrée = `10` $\Rightarrow$ sortie 3 activée (0100)	Sélection = `10` $\Rightarrow$ entrée 3 transmise à la sortie

TABLE 1 – Comparaison fonctionnelle entre un Décodeur et un Multiplexeur

MODULES , TESTBENCH ET SIMULATION

```
module decoder_2to4 (
    /*
    Kambale MARUBA E. (1ICE -EN )
    Bualuti BUKELE E. (1ICE -EE )
    Nkishi DJENGA H. (1ICE -IN )
    */
    input  [1:0] in,
    output [3:0] out
);
    assign out = 4'b0001 << in;
endmodule
```

Code du testbench `decoder_2to4_tb.v`

```
`timescale 1ns/1ps

module decoder_2to4_tb;
 /*
 Kambale MARUBA E. (1ICE -EN )
 Bualuti BUKELE E. (1ICE -EE )
 Nkishi DJENGA H. (1ICE -IN )
 */

    reg  [1:0] in;
    wire [3:0] out;

    decoder_2to4 uut (
        .in(in),
        .out(out)
    );
```

```
    initial begin
        $display("\n=== TEST DECODER 2 TO 4 ===");

        in = 2'b00; #10;
        $display("in = %b => out = %b", in, out);
        if (out !== 4'b0001) $display("ERREUR");

        in = 2'b01; #10;
        $display("in = %b => out = %b", in, out);
        if (out !== 4'b0010) $display("ERREUR");

        in = 2'b10; #10;
        $display("in = %b => out = %b", in, out);
        if (out !== 4'b0100) $display("ERREUR");

        in = 2'b11; #10;
        $display("in = %b => out = %b", in, out);
        if (out !== 4'b1000) $display("ERREUR");

        $display(" -- FIN, REUSSITE --");
        $finish;
    end
endmodule
```

```
PS E:\maruba\computerdesign2025\decoder> iverilog -o sim.out decoder_2to4.v decoder_2to4_tb.v
PS E:\maruba\computerdesign2025\decoder> vvp sim.out

 === TEST DECODER 2 TO 4 ===
 in = 00 => out = 0001
 in = 01 => out = 0010
 in = 10 => out = 0100
 in = 11 => out = 1000
  -- FIN, REUSSITE --
 decoder_2to4_tb.v:39: $finish called at 40000 (1ps)
```

Question 7 : Registre à Décalage (Shift Register)

i) Quel est le principe de fonctionnement d'un registre à décalage ?

Un **registre à décalage** est un circuit séquentiel composé de bascules (flip-flops) connectées en série. Il permet de déplacer (décaler) les bits stockés vers la gauche ou la droite à chaque front d'horloge (*clock*). Ce mécanisme permet de faire transiter des données à travers les différentes cellules du registre, bit par bit, selon une direction donnée.

À chaque impulsion d'horloge :

- Les bits changent de position selon la direction spécifiée.
- Une nouvelle donnée peut être insérée à l'une des extrémités.
- Le bit sortant peut être ignoré, sauvegardé ou réinjecté (cas circulaire).

ii) Quelles sont les différentes configurations d'un registre à décalage ?

Les registres à décalage peuvent être configurés de plusieurs manières selon l'application :

- **Décalage à gauche (Left Shift)** : les bits se déplacent vers la gauche, l'entrée de droite (`din`) reçoit la nouvelle donnée. Exemple :

$$\texttt{q} = 1001 \rightarrow avec \texttt{din} = 1 \Rightarrow \texttt{q}_{suivante} = 0011$$

- **Décalage à droite (Right Shift)** : les bits se déplacent vers la droite, la nouvelle donnée est insérée à gauche.
- **Décalage bidirectionnel (Bidirectional Shift)** : permet de décaler à gauche ou à droite selon un signal de contrôle `dir`.
- **Décalage circulaire (Rotation)** : le bit sortant est réinjecté à l'autre extrémité. Le registre forme une boucle fermée.

iii) Utilisations dans des applications pratiques

Les registres à décalage sont utilisés dans divers domaines de l'électronique et des systèmes numériques :

- **Conversion série-parallèle et parallèle-série** : ils permettent de convertir un flux binaire série en données parallèles, et inversement (ex : transmission UART).
- **Délais numériques** : introduire des délais dans les signaux numériques (retarder une séquence de bits).
- **Multiplication/division par 2** : une opération de décalage correspond à une multiplication ou division par 2.
- **Stockage temporaire** : dans des architectures embarquées, pour tamponner ou aligner les données avant traitement.

MODULES , TESTBENCH ET SIMULATION

1. Décalage à droite

```
module shift_register_right (
    /*
    Kambale MARUBA E. (1ICE -EN )
    Bualuti BUKELE E. (1ICE -EE )
    Nkishi DJENGA H. (1ICE -IN )
    */

    input        clk,
    input        rst,
    input        enable,
    input        din,
    output [3:0] q
);
    reg [3:0] data;
    always @(posedge clk or posedge rst) begin
        if (rst)
            data <= 4'b0000;
        else if (enable)
            data <= {din, data[3:1]}; // D calage    droite
    end
    assign q = data;
endmodule
```

Code du testbench `shift_register_right_tb.v`

```
`timescale 1ns/1ps

module shift_register_right_tb;
 /*
 Kambale MARUBA E. (1ICE -EN )
 Bualuti BUKELE E. (1ICE -EE )
 Nkishi DJENGA H. (1ICE -IN )
 */

    reg clk, rst, enable, din;
    wire [3:0] q;

    shift_register_right uut (
        .clk(clk), .rst(rst), .enable(enable), .din(din), .q(q)
    );

    initial clk = 0;
    always #5 clk = ~clk;

    initial begin
        $display("\n=== SHIFT REGISTER DROITE ===");

        rst = 1; enable = 0; din = 0; #10;
        rst = 0;
```

```
        enable = 1;
        din = 1; #10;
        din = 0; #10;
        din = 1; #10;
        din = 1; #10;

        enable = 0; #10;
        $finish;
    end

    initial begin
        $monitor("Time=%0t | din=%b | q=%b", $time, din, q);
    end
endmodule
```

```
PS E:\maruba\computerdesign2025\shift_register\droite> iverilog -o sim.out shift_register_right.v shift_register_right_tb.v
PS E:\maruba\computerdesign2025\shift_register\droite> vvp sim.out

=== SHIFT REGISTER DROITE ===
Time=0 | din=0 | q=0000
Time=10000 | din=1 | q=0000
Time=15000 | din=1 | q=1000
Time=20000 | din=0 | q=1000
Time=25000 | din=0 | q=0100
Time=30000 | din=1 | q=0100
Time=35000 | din=1 | q=1010
Time=45000 | din=1 | q=1101
shift_register_right_tb.v:34: $finish called at 60000 (1ps)
PS E:\maruba\computerdesign2025\shift_register\droite>
```

2. Décalage à gauche

```
module shift_register_left (
    /*
    Kambale MARUBA E. (1ICE -EN )
    Bualuti BUKELE E. (1ICE -EE )
    Nkishi DJENGA H. (1ICE -IN )
    */

    input        clk,
    input        rst,
    input        enable,
    input        din,
    output [3:0] q
);
    reg [3:0] data;
    always @(posedge clk or posedge rst) begin
        if (rst)
            data <= 4'b0000;
        else if (enable)
            data <= {data[2:0], din}; // D calage    gauche
    end
    assign q = data;
endmodule
```

Code du testbench `shift_register_left_tb.v`

```
`timescale 1ns/1ps

module shift_register_left_tb;
 /*
 Kambale MARUBA E. (1ICE -EN )
 Bualuti BUKELE E. (1ICE -EE )
 Nkishi DJENGA H. (1ICE -IN )
 */

    reg clk, rst, enable, din;
    wire [3:0] q;

    shift_register_left uut (
        .clk(clk), .rst(rst), .enable(enable), .din(din), .q(q)
    );

    initial clk = 0;
    always #5 clk = ~clk;

    initial begin
        $display("\n=== SHIFT REGISTER GAUCHE ===");

        rst = 1; enable = 0; din = 0; #10;
        rst = 0;

        enable = 1;
        din = 1; #10;
        din = 0; #10;
        din = 1; #10;
        din = 1; #10;

        enable = 0; #10;
        $finish;
    end

    initial begin
        $monitor("Time=%0t | din=%b | q=%b", $time, din, q);
    end
endmodule
```

```
PS E:\maruba\computerdesign2025\shift_register\gauche> iverilog -o sim.out shift_register_left.v shift_register_left_tb.v
PS E:\maruba\computerdesign2025\shift_register\gauche> vvp sim.out

=== SHIFT REGISTER GAUCHE ===
Time=0 | din=0 | q=0000
Time=10000 | din=1 | q=0000
Time=15000 | din=1 | q=0001
Time=20000 | din=0 | q=0001
Time=25000 | din=0 | q=0010
Time=30000 | din=1 | q=0010
Time=35000 | din=1 | q=0101
Time=45000 | din=1 | q=1011
shift_register_left_tb.v:34: $finish called at 60000 (1ps)
```

CONCLUSION

À travers ce travail pratique, nous avons approfondi notre compréhension des composants logiques fondamentaux utilisés dans la conception des architectures matérielles. L'étude de l'additionneur binaire nous a permis d'analyser les mécanismes d'addition multi-bits avec gestion de retenue, tandis que le décodeur 2 à 4 nous a permis d'explorer les principes d'adressage et de sélection dans un système numérique. Enfin, le registre à décalage a illustré l'importance du traitement séquentiel des données dans les circuits embarqués modernes.

La mise en œuvre des circuits en langage **Verilog HDL** et leur validation par des *bancs de test* ont renforcé notre capacité à traduire les concepts théoriques en applications concrètes. Cette démarche contribue significativement à notre formation d'ingénieur, en nous préparant à concevoir et analyser des systèmes numériques fiables, modulaires et performants.

Nous remercions nos encadreurs pour leur accompagnement et leur expertise, ainsi que l'ensemble du corps professoral pour les compétences acquises tout au long de ce cours.