Computer Architecture: Calculator Design **Fall 2024**

Homework 8 — Novmember 8

Lecturer: Hu Weiwu *Completed by: 2022K8009929010 Zhang Jiawei*

8.1

(1) $101011_2 = 43_{10}$

$001101_2 = 13_{10}$

$01011010_2 = 90_{10}$

$0000111010000101_2 = 3717_{10}$

(2) $42_{10} = 101010_2$

$79_{10} = 1001111_2$

$811_{10} = 1100101011_2$

$374_{10} = 101110110_2$

(3) $8AE_{16} = 2222_{10}$

$C18D_{16} = 49549_{10}$

$B379_{16} = 45945_{10}$

$100_{16} = 256_{10}$

(4) $81783_{10} = 13F77_{16}$

$1922_{10} = 782_{16}$

$345208_{10} = 54478_{16}$

$5756_{10} = 167C_{16}$

8.2

32 位无符号二进制数表示范围是 $0 \sim 2^{32} - 1$;
32 位二进制原码表示范围是 $-(2^{31} - 1) \sim 2^{31} - 1$;
32 位二进制补码表示范围是 $-2^{31} \sim 2^{31} - 1$。

8.3

$[45_{10}]_{原} = 00101101$, $[45_{10}]_{补} = 00101101$;
$[-59_{10}]_{原} = 10111011$, $[-59_{10}]_{补} = 11000101$;
$[-128_{10}]_{原}$ 溢出, $[-128_{10}]_{补} = 10000000$;
$[119_{10}]_{原} = 01110111$, $[119_{10}]_{补} = 01110111$;
$[127_{10}]_{原} = 01111111$, $[127_{10}]_{补} = 01111111$。
$[128_{10}]_{原}$ 溢出, $[128_{10}]_{补}$ 溢出;

$[0]_{原} = 00000000$, $[0]_{补} = 00000000$;

$[-1_{10}]_{原} = 10000001$, $[-1_{10}]_{补} = 11111111$。

8.4

00101100_2 作为原码扩展到 16 位为 0000000000101100，作为补码扩展到 16 位为 0000000000101100;

11010100_2 作为原码扩展到 16 位为 1000000001010100，作为补码扩展到 16 位为 1111111110101100;

10000001_2 作为原码扩展到 16 位为 1000000000000001，作为补码扩展到 16 位为 1111111110000001;

00010111_2 作为原码扩展到 16 位为 0000000000010111，作为补码扩展到 16 位为 0000000000010111。

8.5

(1) 0 转化为单精度浮点数为 0x00000000;

116.25 转化为单精度浮点数为 0x42E88000;

-4.375 转化为单精度浮点数为 0xC08C0000。

(2) -0 转化为双精度浮点数为 0x8000000000000000;

116.25 转化为双精度浮点数为 0x405D100000000000;

-2049.5 转化为双精度浮点数为 0xC0A0030000000000。

(3) 0xff800000 转化为十进制数为负无穷大;

0x7fe00000 转化为十进制数为非数;

(4) 0x8008000000000000 转化为二进制数为 -0.1×2^{-1023}，十进制数为 -0.5×2^{-1023};

0x7065020000000000 转化为二进制数为 $1.01010000001 \times 2^{775}$，十进制数为 $1.31298828125 \times 2^{775}$。

8.6

逻辑电路表达式为 $Y = \overline{(A+B)C}$.

8.7

真值表如下：

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

8.8

$Y = A \oplus B = A\overline{B} + \overline{A}B = \overline{\overline{A\overline{B}} \cdot \overline{\overline{A}B}}$.

画出电路图如下：

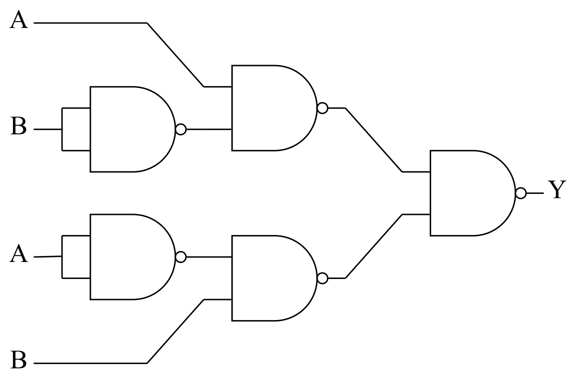

图 **8.1.** 二输入 XOR 电路

8.9

搭建出的的电路如下：

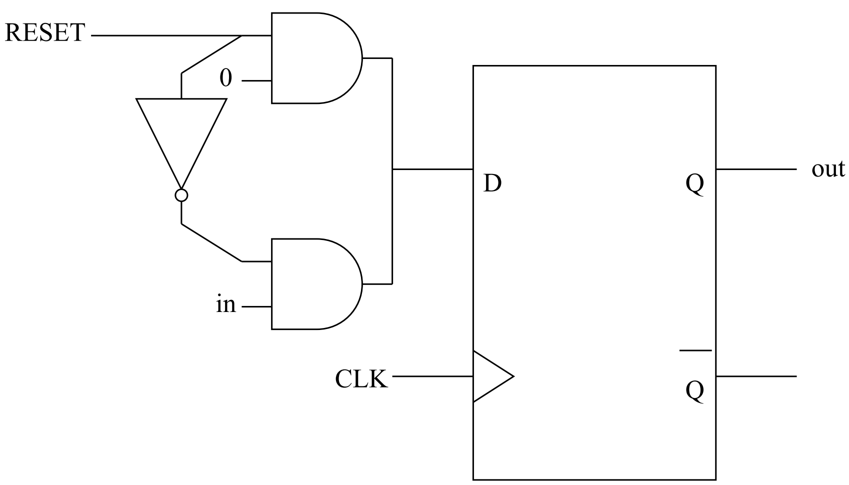

图 **8.2.** 具有同步复位的 D 触发器

8.10

$[X]_{补} + [Y]_{补} = 2^n + X + 2^n + Y \pmod{2^n} = 2^n + X + Y = [X+Y]_{补}$。

8.11

只需将 Y 替换为-Y 即可证明。

8.12

(1) 32 位行波进位加法器的延迟为 $3T + 31 \times 2T = 65T$。

(2) 由于每个逻辑门扇入不超过 4 个，故块内输出进位延迟为 $4T$，故输出所有进位总延迟为 $32T$；最高位产生结果延迟为 $7 \times 4T + 2T + 3T = 33T$，再加上产生 p_i, g_i 的延迟 $2T$，故总延迟为 $35T$。

(3) 生成第一级 p_i, g_i 需要 $2T$，生成所有进位输出需要 $10T$，再通过 $3T$ 算出最终结果，故总延迟为 $15T$。

8.13

资源有限/要求低功耗/速度要求不高/面积较小时，可以采用行波进位加法器。

8.14

以图 8-22 所示 4 个 16 位块内并行块间并行加法器进行层次化扩展，将其 P 和 G 输出作为更高一层的进位输入，最终得到一个 64 位加法器。下层的 4 块 16 位先行进位逻辑根据各块所对应的 p_i 和 g_i 生成各自的块间进位生成因子 G 和块间进位传递因子 P; 上层的 4 位先行进位逻辑把下层的先行进位逻辑生成的 P 和 G 作为本层的 p_i 和 g_i 输入, 生成块间的进位 c_{16}、c_{32} 和 c_{48}; 下层的每块 4 位先行进位逻辑分别把 c_0 以及上层计算出的 c_{16}、c_{32} 和 c_{48} 作为各自块的进位输入 c_0，再结合本地的 p_i 和 g_i 分别计算出本块内部所需要的每一位进位。

8.15

令 $X = -3, Y = 7$，则 $[X \times Y]_{补} = [-21]_{补} = 101011$；而 $[X]_{补} \times [Y]_{补} = 1101 \times 0111 = 1011011$。有 $[X \times Y]_{补} \neq [X]_{补} \times [Y]_{补}$。

8.16

$[X \times 2^n]_{补} = X \times 2^n + 2^m \pmod{2^m} = X \times 2^n + 2^n \times 2^m = (X + 2^m) \times 2^n = [X]_{补} \times 2^n$。

8.17

(1) 使用 16 位块内并行块间并行加法器搭建加法树，第一级生成 p_i, g_i 需要 $2T$，生成所有进位输出需要 $6T$，再通过 $3T$ 算出最终结果，故总延迟为 $11T$，两级加法树总延迟为 $22T$。

(2) 华莱士树将 4 个 16 位加数转化成 2 个 16 位加数，延迟为 $6T$，再通过先行进位加法器计算出最终结果，延迟为 $11T$，故总延迟为 $17T$。

8.18

两位 Booth 补码乘法器中，$[-X]_{补}$ 是对 $-X$ 取补码，$[-2X]_{补}$ 是先将 $-X$ 左移一位，再取补码，然后把它们加到原来的部分积上；华莱士树补码乘法器中，补码的获取方式是同样的，将多个部分积压缩成两个部分积，然后通过先行进位加法器计算出最终结果。

8.19

```
module booth_multiplier (
   input signed [31:0] A, // 32位乘数
   input signed [31:0] B, // 32位被乘数
   output signed [63:0] result // 64位乘积
);

   wire signed [32:0] pp[15:0]; // 存储16个部分积 (32+1位用于扩展符号位)
   integer i;

   always @(*) begin
      pp[0] = booth_encode(A, {B[1:0], 1'b0}); // Booth 编码生成每一组的部分积
      for (i = 1; i < 16; i = i + 1) begin
         pp[i] = booth_encode(A, B[2*i+1:2*i-1]);
```

```
      end
   end

   wire [63:0] sum, carry; // 用于华莱士树中的部分积和进位
   wallace_tree wt(.pp(pp), .sum(sum), .carry(carry));

   assign result = sum + carry;

endmodule

// Booth Encoding Function (2位一乘)
function signed [32:0] booth_encode;
   input signed [31:0] A;
   input [2:0] B; // 2位一乘，B为编码位 + 上一位
   case (B)
      3'b001, 3'b010: booth_encode = A; // +X
      3'b011:     booth_encode = A << 1; // +2X
      3'b100:     booth_encode = -(A << 1); // -2X
      3'b101, 3'b110: booth_encode = -A; // -X
      default:    booth_encode = 0;   // 0
   endcase
endfunction

module wallace_tree (
   input [63:0] pp[15:0], // 输入16个部分积，每个64位宽
   output [63:0] sum, // 华莱士树的部分和输出
   output [63:0] carry // 华莱士树的进位输出
);

   wire [63:0] layer_sum[5:0]; // 中间层的和
   wire [63:0] layer_carry[5:0]; // 中间层的进位
   integer i;

   // 第 1 层: 压缩 16 个部分积
   // 每一列使用全加器/半加器来处理部分积的压缩
   generate
      for (i = 0; i < 64; i = i + 1) begin
         // 处理每一列的压缩: 三输入全加器和二输入半加器
         if (i < 16) begin
            // 使用全加器压缩每列
            full_adder fa1(pp[0][i], pp[1][i], pp[2][i], layer_sum[0][i],
                layer_carry[0][i]);
```

```
            full_adder fa2(pp[3][i], pp[4][i], pp[5][i], layer_sum[1][i],
                layer_carry[1][i]);
            // 更多的全加器操作
        end
    end
endgenerate

// 后续层依次压缩输出
generate
    for (i = 1; i < 5; i = i + 1) begin
        // 将每层产生的部分和与进位继续压缩
        for (j = 0; j < 64; j = j + 1) begin
            full_adder fa(layer_sum[i-1][j], layer_carry[i-1][j], pp[j+1],
                layer_sum[i][j], layer_carry[i][j]);
        end
    end
endgenerate

// 最后一层的和、进位输出
assign sum = layer_sum[4];
assign carry = layer_carry[4];

endmodule

// 定义全加器模块
module full_adder(
    input a, b, cin,
    output sum, cout
);
    assign sum = a ^ b ^ cin;
    assign cout = (a & b) | (b & cin) | (cin & a);
endmodule
```

8.20

不可以。因为 π 是无限不循环小数，无法用有限的位数表示。单精度浮点数的小数部分只有 23 位，双精度浮点数的小数部分只有 52 位，故无法精确表示 π。