# 实验九—加法器 乘法器

姓名：张云策 学号：3200105787 学院：计算机科学与技术学院

课程名称：计算机系统 1 同组学生姓名：/

实验时间：2021.6.6 实验地点：紫金港东 4-509 指导老师：常瑞

# 一、 实验目的和要求

- 完成一个 32bit 的加法器
- 完成一个 32bit 的乘法器

# 二、 实验内容和原理

## 2.1 实验内容

使用 FPGA 开发板以及 Verilog 语言设计实现了 32bit 加法器以及 32bit 乘法器。

## 2.2 设计模块

- 加法器：
  Ripple-carrry adder：
  1 bit 加法器：

```
module bit_adder
(
        input Ai,
        input Bi,
        input c_in,
        output sum,
        output c_out
);
assign sum=Ai^Bi^c_in;
assign c_out=(Ai&Bi)|(Bi&c_in)|(Ai&c_in);
endmodule
```

  由 4 个 1 bit 加法器组成的四位加法器：

```
module ripple_carrt_adder(
            input [3:0] Ai,
            input [3:0] Bi,
            input c_in,
            output [3:0] sum,
            output c_out);
            wire [4:0]c;
            assign c[0] = c_in;
             assign c_out = c[4];
             bit_adder adder1(Ai[0],Bi[0],c[0],sum[0],c[1]);
             bit_adder adder2(Ai[1],Bi[1],c[1],sum[1],c[2]);
             bit_adder adder3(Ai[2],Bi[2],c[2],sum[2],c[3]);
             bit_adder adder4(Ai[3],Bi[3],c[3],sum[3],c[4]);
             endmodule
```

由 8 个四位加法器组成的 32 位加法器：

```
//串行进位加法器
module Adder(
    input [31:0] Ai,
    input [31:0] Bi,
    input c_in,
    output [31:0] SUM,
    output c_out
    );
wire [7:0] c;
assign c[0] = c_in;
assign c_out = c[8];
ripple_carrt_adder adder1 (Ai[3:0], Bi[3:0], c[0], SUM[3:0], c[1]);
ripple_carrt_adder adder2 (Ai[7:4], Bi[7:4], c[1], SUM[7:4], c[2]);
ripple_carrt_adder adder3 (Ai[11:8], Bi[11:8], c[2], SUM[11:8], c[3]);
ripple_carrt_adder adder4 (Ai[15:12], Bi[15:12], c[3], SUM[15:12], c[4]);
ripple_carrt_adder adder5 (Ai[19:16], Bi[19:16], c[4], SUM[19:16], c[5]);
ripple_carrt_adder adder6 (Ai[23:20], Bi[23:20], c[5], SUM[23:20], c[6]);
ripple_carrt_adder adder7 (Ai[27:24], Bi[27:24], c[6], SUM[27:24], c[7]);
ripple_carrt_adder adder8 (Ai[31:28], Bi[31:28], c[7], SUM[31:28], c[8]);
endmodule
```

Carry-lookahead adder：

4-bit carry-lookahead adder:

```
module carry_lookahead_adder(
          input signed [3:0] a,
          input signed [3:0] b,
          input c_in,
          output signed [3:0] sum,
          output c_out);

      wire [3:0] G;
      wire [3:0] P;

      assign G = a&b;
      assign P = a^b;

      wire [4:0] c;
      assign c[0] = c_in;
      assign c[1] = G[0]|(P[0]&c_in);
      assign c[2] = G[1]|(P[1]&G[0])|(P[1]&P[0]&c_in);
      assign c[3] = G[2]|(P[2]&G[1])|(P[2]&P[1]&G[0])|(P[2]&P[1]&P[0]&c_in);
      assign c[4] = G[3]|(P[3]&G[2])|(P[3]&P[2]&G[1])|(P[3]&P[2]&P[1]&G[0])|(P[3]&P[2]&P[1]&P[0]&c_in);
      assign sum = a^b^c[3:0];
      assign c_out = c[4];

endmodule
```

32-bit carry-lookahead adder consist of 8 4-bit carry lookahead adder:

```
module Adder(
    input signed [31:0] Ai, Bi,
    input c_in,
    output signed [31:0] SUM,
    output c_out
    );

wire [8:0] c;
assign c[0] = c_in;
assign c_out = c[8];
carry_lookahead_adder adder1(Ai[3:0], Bi[3:0], c[0], SUM[3:0], c[1]);
carry_lookahead_adder adder2(Ai[7:4], Bi[7:4], c[1], SUM[7:4], c[2]);
carry_lookahead_adder adder3(Ai[11:8], Bi[11:8], c[2], SUM[11:8], c[3]);
carry_lookahead_adder adder4(Ai[15:12], Bi[15:12], c[3], SUM[15:12], c[4]);
carry_lookahead_adder adder5(Ai[19:16], Bi[19:16], c[4], SUM[19:16], c[5]);
carry_lookahead_adder adder6(Ai[23:20], Bi[23:20], c[5], SUM[23:20], c[6]);
carry_lookahead_adder adder7(Ai[27:24], Bi[27:24], c[6], SUM[27:24], c[7]);
carry_lookahead_adder adder8(Ai[31:28], Bi[31:28], c[7], SUM[31:28], c[8]);
endmodule
```

Carry_lookahead adder 的先进性在于它可以利用进位直接进行计算，提前得出数值，不同于串行加法器需要按次序进行，从而减少加法计算的次数，达到节约时间的目的。

- 乘法器：

普通乘法器：

```
//普通乘法器
module normal_mul(
    input  [31:0] Ai,
    input  [31:0] Bi,
    output reg [63:0] MUL
    );
    reg [63:0]_a;
    reg [30:0]_b;
    reg [63:0]_mul;
    reg sign;
    always@(Ai or Bi)
    begin
        _mul = 0;
        _a = {32'd0,Ai[31]?(~Ai+1):Ai};
        _b = Bi[31]?(~Bi+1):Bi;
        sign = Ai[31]^Bi[31];
        repeat(32)
        begin
            if(_b[0]) _mul = _mul + _a;
            _a = _a << 1;
            _b = _b>> 1;
        end
         MUL = (sign)?(~_mul+1):_mul;
    end
endmodule
```

Booth 乘法器：

```verilog
module Booth_mul
(
    input   clk,
    input   rst,
    input   start,
    input  [ 31 : 0 ]     Ai,
    input  [ 31: 0 ]     Bi,
    output [ 63 : 0 ]   MUL,
    output Done
);
reg [ 31: 0 ] i;
reg [ 64: 0 ] P;
reg [ 31: 0 ] A_reg;
reg [ 31: 0 ] A_bm;
reg [ 31: 0 ] N;
reg isDone;
always @ ( posedge clk or negedge rst )
begin
    if (!rst)
    begin
        i <= 0;
        P <= 0;
        A_reg <= 0;
        A_bm <= 0;
        N <=0;
        isDone <= 0;
    end
    else if (start)
    begin
        case (i)

            0:
                begin
                    A_reg <= Ai;
                    A_bm <= ~Ai + 1'b1;
                    P <= { 8'd0, Bi, 1'b0 };
                    i <= i + 1'b1;
                    N <= 0;
                end
            1:
                begin
                    if (N == 32)
                        begin
                            N <= 0;
                            i <= i + 2'b10;
                        end
                    else if (P[1:0] == 2'b00 | P[1:0] == 2'b11)
                        begin
                            P <= P;
                            i <= i + 1'b1;
                        end
                    else if (P[1:0] == 2'b01)
                        begin
                            P <= {P[64:32] + A_reg,P[31:0]};
                            i <= i + 1'b1;
                        end
                    else if (P[1:0] == 2'b10)
                        begin
                            P <= {P[64:32] + A_bm,P[31:0]};
                            i <= i + 1'b1;
                        end
                end
            2:
                begin
                    P <= {P[64],P[64:1]};
                    N <= N + 1'b1;
                    i <= i - 1'b1;
                end
            3:
                begin
                    isDone <= 1;
                    i <= i + 1'b1;
                end
            4:
                begin
                    isDone <= 0;
                    i <= 0;
                end

        endcase
    end
end
assign Done = isDone;
assign MUL = P[64:1];
endmodule
```

Testbench：（以普通加法器为例）

```
normal_mul normal_t
   (
  .Ai(Ai),
  .Bi(Bi),
  .MUL(MUL)
   );
  initial
  begin
  Ai = 32'b0;
  Bi = 32'b0;
  #50
   Ai = 32'b 01001011011;   Bi = 32'b 11111111110;
   #50
   Ai = 32'h 23122ff2;   Bi = 32'h 21323211;
   #50
   Ai = 32'h FFFFFFFF;   Bi = 32'b 010101111100101;
   #50
   Ai = 32'b 0111010100101001010l;   Bi = 32'b 010101011011111;
   #50
   Ai = 32'b 0110011101010100101;   Bi = 32'b 0101111111101;
    #50
   Ai = 32'b 0111010101001010011;   Bi = 32'b 0101111111101;
    #50
   Ai = 32'b 0111010101011100101;   Bi = 32'b 0101111111101;
    #50
   Ai = 32'b 01111010101010100101;   Bi = 32'b 0101111111101;
    #50
   Ai = 32'b 0111010101001010l;   Bi = 32'b 0101010010101001101;
```

Booth 算法的原理是先判断 P[1:0]，然后操作 P 空间，最后 P 空间移位，根据 P 空间的最高位来给移位后的 P 选择补 1 or 补 0，直接操作位运算，从而达到提高效率的目的。

# 三、 主要仪器设备

Vivado
FPGA

# 四、 操作方法与实验步骤

## 4.1 操作方法

加法器：

- SW[7:5]控制了显示的通道。SW[7:5]不同的值对应的通道的显示内容为

0. Ai
1. Bi
2. 加法器的结果
3. 乘法器的结果（不显示）

- 当 SW[15]=1 时，可以使用按键修改 Ai 和 Bi 的值。将 SW[15]置为高，然后调整 SW[7:5]到对应的 Ai 的通道（或者 Bi）。此时可以看见调整的数字正在闪烁。使用左或者右按键可以选择调整的数字，使用上和下键对数字做加一或者减一。

乘法器：

- SW[7:5]控制了显示的通道。SW[7:5]不同的值对应的通道的显示内容为

0. Ai
1. Bi
2. 乘法器的结果（前 32 位）
3. 乘法器的结果（后 32 位）

- 当 SW[15]=1 时，可以使用按键修改 Ai 和 Bi 的值。将 SW[15]置为高，然后调整 SW[7:5]到对应的 Ai 的通道（或者 Bi）。此时可以看见调整的数字正在闪烁。使用左或者右按键可以选择调整的数字，使用上和下键对数字做加一或者减一。

## 4.2 实验步骤

- 加法器：

  1. 行波进位加法器：

  核心模块：

  1-bit 加法器

  Code：

```
module bit_adder
    (
        input Ai,
        input Bi,
        input c_in,
```

```
            output sum,
            output c_out
        );
        assign sum=Ai^Bi^c_in;
        assign c_out=(Ai&Bi)|(Bi&c_in)|(Ai&c_in);
endmodule
```

组合部分代码：

4-bit：

```
input [3:0] Ai,
                input [3:0] Bi,
                input c_in,
                output [3:0] sum,
                output c_out);
                wire [4:0]c;
                assign c[0] = c_in;
                  assign c_out = c[4];
                  bit_adder adder1(Ai[0],Bi[0],c[0],sum[0],c[1]);
                  bit_adder adder2(Ai[1],Bi[1],c[1],sum[1],c[2]);
                  bit_adder adder3(Ai[2],Bi[2],c[2],sum[2],c[3]);
                  bit_adder adder4(Ai[3],Bi[3],c[3],sum[3],c[4]);
```

32-bit：

```
        input [31:0] Ai,
        input [31:0] Bi,
        input c_in,
        output [31:0] SUM,
        output c_out
        );
    wire [7:0] c;
    assign c[0] = c_in;
    assign c_out = c[8];
    ripple_carrt_adder adder1 (Ai[3:0], Bi[3:0], c[0], SUM[3:0], c[1]);
    ripple_carrt_adder adder2 (Ai[7:4], Bi[7:4], c[1], SUM[7:4], c[2]);
    ripple_carrt_adder adder3 (Ai[11:8], Bi[11:8], c[2], SUM[11:8], c[3]);
    ripple_carrt_adder adder4 (Ai[15:12], Bi[15:12], c[3], SUM[15:12], c[4]);
    ripple_carrt_adder adder5 (Ai[19:16], Bi[19:16], c[4], SUM[19:16], c[5]);
    ripple_carrt_adder adder6 (Ai[23:20], Bi[23:20], c[5], SUM[23:20], c[6]);
    ripple_carrt_adder adder7 (Ai[27:24], Bi[27:24], c[6], SUM[27:24], c[7]);
    ripple_carrt_adder adder8 (Ai[31:28], Bi[31:28], c[7], SUM[31:28], c[8]);
```

超前进位加法器：

核心模块：

4-bit 超前进位：

```
module carry_lookahead_adder(
            input signed [3:0] a,
            input signed [3:0] b,
```

```
        input c_in,
        output signed [3:0] sum,
        output c_out);

    wire [3:0] G;
    wire [3:0] P;

      assign G = a&b;
      assign P = a^b;

    wire [4:0] c;
    assign c[0] = c_in;
    assign c[1] = G[0]|(P[0]&c_in);
    assign c[2] = G[1]|(P[1]&G[0])|(P[1]&P[0]&c_in);
    assign c[3] = G[2]|(P[2]&G[1])|(P[2]&P[1]&G[0])|(P[2]&P[1]&P[0]&c_in);
    assign c[4] = G[3]|(P[3]&G[2])|(P[3]&P[2]&G[1])|(P[3]&P[2]&P[1]&G[0])|(P[3]&P[2]&P[1]&P[0]&c_in);
    assign sum = a^b^c[3:0];
    assign c_out = c[4];
```

- 乘法器：

  Normal multiplier：

  通过循环，实现乘法运算：

```
    reg [63:0]A;
    reg [30:0]B;
    reg [63:0]mul;
    reg sign;
    always@(Ai or Bi)
    begin
        mul = 0;
        A = {32'd0,Ai[31]?(~Ai+1):Ai};
        B = Bi[31]?(~Bi+1):Bi;
        sign = Ai[31]^Bi[31];
        repeat(32)
        begin
            if(B[0]) mul = mul + A;
            A = A << 1;
            B =B>> 1;
        end
          MUL = (sign)?(~mul+1):mul;
end
```

  Booth multiplier:

  与普通乘法器的主要区别在于判断“11、10、01、00”的条件，进而实现 booth 算法。

  Code：

```
always @ ( posedge clk or negedge rst )
```

```
begin
    if (!rst)
    begin
        i <= 0;
        P <= 0;
        A_reg <= 0;
        A_bm <= 0;
        N <=0;
        isDone <= 0;
    end
    else if (start)
    begin
        case (i)//判断“00、11、10、01”
            0:
                begin
                    A_reg <= Ai;
                    A_bm <= ~Ai + 1'b1;
                    P <= { 8'd0, Bi, 1'b0 };
                    i <= i + 1'b1;
                    N <= 0;
                end
            1:
                begin
                    if (N == 8)
                        begin
                            N <= 0;
                            i <= i + 2'b10;
                        end
                    else if (P[1:0] == 2'b00 | P[1:0] == 2'b11)
                        begin
                            P <= P;
                            i <= i + 1'b1;
                        end
                    else if (P[1:0] == 2'b01)
                        begin
                            P <= {P[64:32] + A_reg,P[31:0]};
                            i <= i + 1'b1;
                        end
                    else if (P[1:0] == 2'b10)
                        begin
                            P <= {P[64:32] + A_bm,P[31:0]};
                            i <= i + 1'b1;
                        end
```

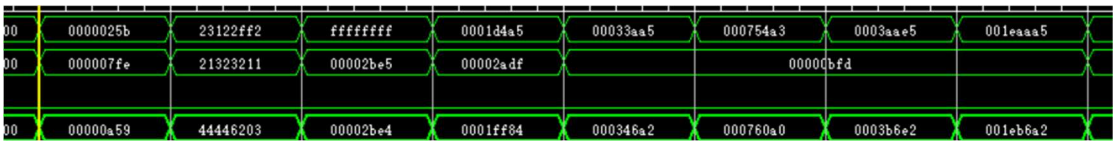
```
                end
            2:
                begin
                    P <= {P[64],P[64:1]};
                    N <= N + 1'b1;
                    i <= i - 1'b1;
                end
            3:
                begin
                    isDone <= 1;
                    i <= i + 1'b1;
                end
            4:
                begin
                    isDone <= 0;
                    i <= 0;
                end

        endcase
    end
end

assign Done = isDone;
assign MUL = P[64:1];//舍弃最高位最低位
endmodule
```

# 五、 实验结果与分析

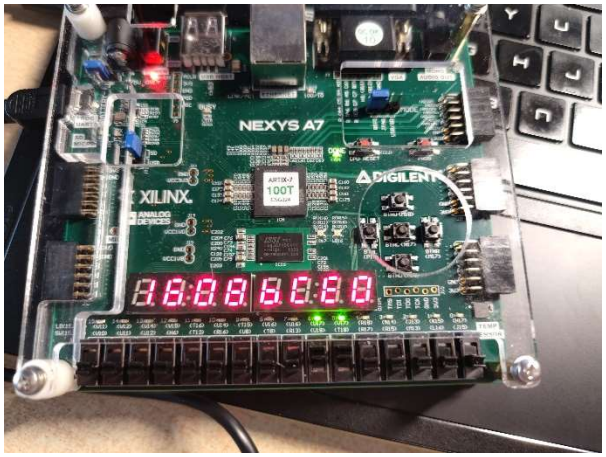
加法器：
仿真激励代码：
两种加法器很相似，故只举一例

```
Adder adder_t
(
  .Ai(Ai),
  .Bi(Bi),
  .c_in(c_in),
  .SUM(SUM)
);
  initial
  begin
  Ai = 32'b0;
  Bi = 32'b0;
  c_in = 1'b0;
  #50
   Ai = 32'b 01001011011;   Bi = 32'b 11111111110;
   #50
   Ai = 32'h 23122ff2;   Bi = 32'h 21323211;
   #50
   Ai = 32'h FFFFFFFF;   Bi = 32'b 010101111100101;
   #50
   Ai = 32'b 01110101001010010l;   Bi = 32'b 010101011011111;
   #50
   Ai = 32'b 01100111010101001O1;   Bi = 32'b 0101111111101;
    #50
   Ai = 32'b 01110101010010001l;   Bi = 32'b 0101111111101;
    #50
   Ai = 32'b 01110101010111001O1;   Bi = 32'b 0101111111101;
    #50
   Ai = 32'b 011110101010101001O1;   Bi = 32'b 0101111111101;
```

波形图：




上板实例：



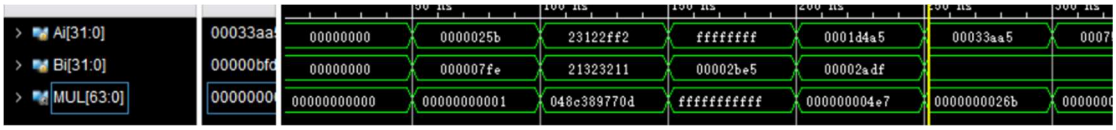

乘法器：

仿真激励代码：

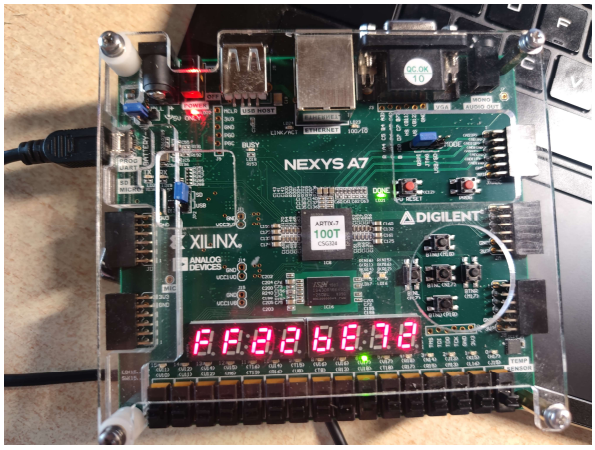
```
normal_mul normal_t
   (
   .Ai(Ai),
   .Bi(Bi),
   .MUL(MUL)
    );
   initial
   begin
   Ai = 32'b0;
   Bi = 32'b0;
   #50
    Ai = 32'b 01001011011;   Bi = 32'b 11111111110;
    #50
    Ai = 32'h 23122ff2;   Bi = 32'h 21323211;
    #50
    Ai = 32'h FFFFFFFF;   Bi = 32'b 010101111100101;
    #50
    Ai = 32'b 011101010010100101;   Bi = 32'b 010101011011111;
    #50
    Ai = 32'b 0110011101010100101;   Bi = 32'b 0101111111101;
     #50
    Ai = 32'b 01110101010010100011;   Bi = 32'b 0101111111101;
     #50
    Ai = 32'b 0111010101011100101;   Bi = 32'b 0101111111101;
     #50
    Ai = 32'b 01111010101010100101;   Bi = 32'b 0101111111101;
     #50
    Ai = 32'b 011101010100101;   Bi = 32'b 01010100101001101;
```

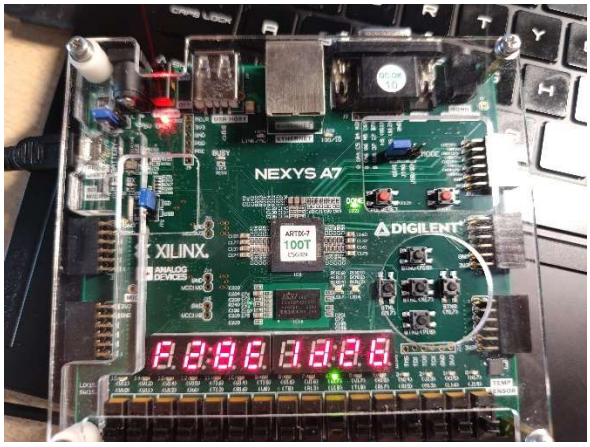
```
booth_mul booth_t
  (
  .Ai(Ai),
  .Bi(Bi),
  .MUL(MUL),
  .clk(clk),
  .rst(rst)
   );
  initial
  begin
  Ai = 32'b0;
  Bi = 32'b0;
  #50
   Ai = 32'b 01001011011;   Bi = 32'b 11111111110;
   #50
   Ai = 32'h 23122ff2;   Bi = 32'h 21323211;
   #50
   Ai = 32'h FFFFFFFF;   Bi = 32'b 010101111100101;
   #50
   Ai = 32'b 011101010010100101;   Bi = 32'b 010101011011111;
   #50
   Ai = 32'b 01100111010101001 01;   Bi = 32'b 0101111111101;
```

波形图：




上板实例：



# 六、 讨论、心得

通过这次实验，能够更好了解 Verilog 和 FPGA 的运用，产生了一些兴趣，但同时也对下周的指令集设计有点恐惧了，希望到时候会做吧（逃