实验报告——自动售货机

备注：该文档中一些截图可能不清晰，因此在提交的压缩包里面同时提供了独立于文档的截图，便于助教查看

一、需求分析

下面列出的是针对文档中没有明确规定的需求，并给出我的选择方案：

1. 因为实验要求中并不要求自动售货机有找零功能，因此为了简单起见，在具体实验中，我并没有考虑同时买了两件及以上商品的情况；因为六种商品中最贵的商品为 13 元，那么我只考虑输入金额最多为 12.5+5=17.5，因此对应已经输入的金额数要用三部分表示，第一部分表示小数点之后的部分，第二部分表示整数，因为四个比特最大表示数字为 15，因此需要第三部分来表示最高位整数
2. 文档中没有明确规定商品编号和输入金额是用同一个 7 段数码管显示还是用两个 7 段数码管分别显示，为了使这个简陋的自动售货机更易于使用，在此次实验中，我的选择是用 AN[4]显示选定的商品编号，用 AN[2]、AN[1]和 AN[0]共同组成了输入金额，其中 AN[0]显示的是小数部分，因此整数部分 AN[1]对应的小数点要显示，最高位整数 AN[2]的小数点不需要显示
3. 因为实验中有六种商品，因此用 3 个 switch 表示，那么对应的 000 和 111 都是无效的商品编号，对于这两个无效的商品编号我有三个处理，一是当选定这两个无效编号，对应显示商品编号的 7 段数码管都显示为 0；二是当选定这两个无效编号时，不允许输入货币，也就是按下货币输入的四个按钮无效；三是倘若在购买编号为 1-6 的商品中切换商品编号为 0 或者 7，已经输入的货币不清零
4. 文档里面并没有明确说明当购买一件商品过程中放弃购买时，已经输入的货币是否可以退回，所以我考虑了日常生活中的实际情况，设置了一个 reset 位，用一个 button 来控制，当 reset 被按下时，已经输入的货币清零
5. 文档中并没有提到表示交易完成的等要亮多久，因此我在此次实验中的处理是，当一次交易完成之后，表示此次交易完成的灯到下次交易开始时才会息掉；表示下次交易开始有两种情况：一是改变了商品编号；二是保持当前商品编号不变，重新开始输入货币。因为既改变商品编号，又输入货币按照改变的顺序，一定属于这两种情况的某一种，因此对此种情况，不再分类讨论
6. 通过以上分析，我们可以得出系统的总输入和总输出：
   a) 总输入：时钟、reset、商品编号、当前输入的金额
   b) 总输出：8 个七段数码管的显示、表示当前交易是否完成的 LED 灯的显示

二、实验设计

1.module paid：

a)输入：clk(表示时钟，1-bit)；hasDone(表示此次交易是否完成，1-bit)；reset(表示对已经输入的货币是否清零，1-bit)；No(表示选定的商品编号，3-bit)；money(表示当前输入的货币金额，包括 5 角、1 块、2 块和 5 块，4-bit)

b)**输出：**paid(表示已经输入的货币总额，根据需求分析的第一条，此处选用长度为 5-bit)

c)该模块实现的功能是，保存当前已经输入的货币总数，并且根据商品编号、当前输入的货币金额、当前交易是否完成、是否将已输入的货币清零以及时钟等输入，对保存的输入金额做相应的改变

d)**防抖动处理：**此处的防抖动处理我利用的是文档中给出的方法，因为我利用的是板子上的 E3 引脚，对应的频率是 100MHz，所以我取 count 的最大值为 8000000，每到 clk 的时钟上升沿，就将 count 加一，当 count 达到最大值时，对 clk2 取反，这样 clk2 的时钟周期就降到了 12.5Hz，每到 clk2 的上升沿获取一下按钮状态，就相当于每隔 80ms 获取一次按钮状态，一定程度上避免了在按钮仍然抖动的时候获取按钮的状态而得到无效的值；代码如下：

```
always @(posedge clk) begin
    count = count + 1;
    if(count > 8000000) begin
        clk2 = ~clk2;
        count = 0;
    end
end
```

e)**关键代码解析：**这一模块除了必要的防抖动处理，另外一部分重要的就是利用低频的 clk2 和五个按钮（4 个表示输入金额，1 个表示 reset）对保存已输入金额的寄存器中的值做出对应的改变，首先上代码：

```
always @(posedge clk2 or posedge reset) begin
    if(reset)
        paid = 0;
    else begin
    case(money)
        4'b0001: money_cur = 4'b0001;
        4'b0010: money_cur = 4'b0010;
        4'b0100: money_cur = 4'b0100;
        4'b1000: money_cur = 4'b1010;
        default: money_cur = 0;
    endcase
    if(hasDone && money_cur)
        paid = 0;
    if(No >= 1 && No <= 6)
    paid = paid + money_cur;
    end
end
```

①always 的敏感列表，除了 clk2 的时钟上升沿之外，另外一个就是表示 reset 的上升沿，也就是从 0 变为 1 的时候

②在块的内部，首先判断是否是 reset 触发的，如果是，就将存储已输入金额总数的 paid 寄存器中的值清零

③如果不是，就首先通过模块输入的 money 判断当前的输入金额，也就是 money_cur，因为不考虑多个按钮同时按下的情况，因此在 default 中，多个按钮同时按下，也表示当前输入金额为 0；

④之后，就是最关键的地方，如果当前 hasDone 为 1 并且 money_cur 不为 0,也就是输入了新的金额,那么就将 paid 寄存器清零;这一步的意思是，如果当前交易已经完成了，那么如果接收到了新的非 0 的货币输入，就认为开始了一次新的交易，那么 paid 就要清零

⑤判断 No 是否有效，也就是是否∈[1,6]，如果在，就对 paid 加上当前输入金额数，如果不在，就不处理；意味着，当 No 从一个有效的商品编号切换为 0 或者 7 时，已输入金额不会清零，但是此时不能再投入货币

2.module compDone：

a)**输入**：No(表示当前选定的商品编号，3-bit)，money(表示当前投入的金额，4-bit)，Paid(表示当前已输入的金额总数，6-bit)

b)**输出**：hasDone(表示在已输入金额为 Paid 的情况下，此次交易是否完成)

c)显然，这个模块要通过比较 No 商品的价格和已输入金额 Paid 的值判断此次交易是否完成，因此，每当 No 变化，我们都要获取一下 No 商品的价格 price(5-bit)，便于之后的比较，对于无效的商品编号，取价格为最大，0x1F：

```
always @(No) begin
    case(No)
        3'b001: price = 5'b00001;
        3'b010: price = 5'b00010;
        3'b011: price = 5'b00011;
        3'b100: price = 5'b00100;
        3'b101: price = 5'b01101;
        3'b110: price = 5'b11010;
        default: price = 5'b11111;
    endcase
end
```

d)**关键代码解析**：该模块的作用就是判断当前交易是否完成，输出一个表示是否完成的 hasDone，因此代码实现如下：

```
always @(*) begin
    if(hasDone == 1 || hasDone == 0) begin  end
    else
        hasDone = 0;
    if(hasDone && money_cur)
        hasDone = ~hasDone;
    if(Paid >= price && (No >= 1 || No <= 6))
        hasDone = 1;
end
```

①首先第一部分很巧妙，hasDone 是没有初始值的，在程序刚开始执行时，hasDone 处于不确定状态(x)，为了避免之后取反出问题，我需要在此时对 hasDone 赋一个初始值 0；因为 verilog 中，不确定值无法直接通过是否等于 x 而判断，因此，我判断 hasDone 既不是 0，也不是 1，就认为此时 hasDone 的值是不确定的，赋值为 0

②因为在需求分析中提到了表示交易已完成的 LED 灯亮多长时间的问题，在此次实验中我令 LED 灯一直亮，直到下一次交易开始，对应的代码实现就是当 hasDone=1，也就是交易已完成时，只有新的 money 输入，才认为开始了一次新的交易，改变 hasDone 为 0；对应代码中的 money_cur 与 paid 模块中的意义相同，都是表示当前输入的金额数，保持统一，此处也忽略了多种货币同时输入

③如果已输入金额数 Paid 大于 price，并且 No 是有效的商品编号，认为交易完成，hasDone 赋值为 1

3.module display：

a)输入：clk(表示时钟，1-bit)，hasDone(表示交易是否完成，1-bit)，No(表示选定的商品编号，3-bit)，Paid(表示已输入金额数，6-bit)

b)输出：DoneLED(代表的是表示交易是否完成的 LED，1-bit)，DP(七段数码管对应的小数点，1-bit)，AN(8 个七段数码管，8-bit)，A2G(七段数码管的七位，7-bit)

c)根据七段数码管的原理，此处也要对八个七段数码管做分时显示，和 LAB3 做相同的处理，因为此次需要显示的七段数码管有 8 个（除需求分析第二条提到的四个，其余全是 0），所以 s 取 3-bit：

```
reg [19:0]clkdiv;
wire [2:0]s;
initial begin
    clkdiv = 0;
end
assign s = clkdiv[19:17];
always @(posedge clk) begin
    clkdiv = clkdiv+1;
end
```

d)用 DoneLED 显示交易是否完成，每次 hasDone 变化，就对 DoneLED 重新按照 hasDone 的状态赋值：

```
always @(hasDone) begin
    DoneLED = hasDone;
end
```

e)**获取每一个七段数码管的显示数值**，根据 s 判断：

```
always @(*) begin
    case(s)
        3'b000: digit = Paid[0:0] * 5;
        3'b001: digit = Paid[4:1];
        3'b010: digit = Paid[5:5];
        3'b100: begin if(No <= 6) digit = No; else digit = 0; end
        default: digit = 0;
    endcase
    if(s == 3'b001)
        DP = 0;
    else
        DP = 1;
end
```

①当 s=0 时，显示的小数部分。digit 表示七段数码管显示的数值，因为此处是定点小数，也就是小数点固定的小数，因此，我们约定 Paid 的第 0 位总是表示输入金额的小数部分，为 1 表示 5 角，为 0 表示输入金额恰好为整数，那么对应显示小数位时，要对该值乘 5 得到实际表示的金额数来显示

②当 s=1 时，显示的是低四位的整数部分，s=2 时显示已输入金额的最高位整数部分

③当 s=4 时，显示商品编号，对于无效的商品编号 0 和 7，统一显示为 0，这也就是代码中对 No 是否在[1,6]范围中做分别处理的意义

④小数点，只有 s=1 时，对应位才显示小数点

f)**获取当前显示的是哪一个七段数码管**，也是根据 s 判断的，此处应格外注意是低电平有效：

```
always @(*) begin
    AN = 8'b11111111;
    AN[s] = 0;
end
```

g)**对应每一个七段数码管应该如何显示**，在前两步得到当前应该显示哪一个七段数码管和对应要显示的内容之后，就应该处理如何显示的问题了，此处与上次 LAB 相同，都是根据 digit 的数值选择 a-g 七段分别是否显示：

```
always @(*) begin
  case(digit)
    4'h1: A2G = 7'b1111001;
    4'h2: A2G = 7'b0100100;
    4'h3: A2G = 7'b0110000;
    4'h4: A2G = 7'b0011001;
    4'h5: A2G = 7'b0010010;
    4'h6: A2G = 7'b0000010;
    4'h7: A2G = 7'b1111000;
    4'h8: A2G = 7'b0000000;
    4'h9: A2G = 7'b0010000;
    4'hA: A2G = 7'b0001000;
    4'hB: A2G = 7'b0000011;
    4'hC: A2G = 7'b1000110;
    4'hD: A2G = 7'b0100001;
    4'hE: A2G = 7'b0000110;
    4'hF: A2G = 7'b0001110;
    default: A2G = 7'b1000000;
  endcase
end
```

4.module VEM：

a)**输入：**clk(表示时钟，1-bit，对应 E3 引脚)，
reset(表示已输入金额是否清零，1-bit，对应一个按钮)，
No(表示当前选定的商品编号，3-bit，对应 3 个 switch)，
money(表示当前输入的金额，4-bit，对应 4 个 button)

b)**输出：**DoneLED(显示当前交易是否完成，1-bit，对应一个 LED 灯)，
DP(表示小数点，1-bit)，
AN(对应 8 个七段数码管，8-bit)，
A2G(表示七段数码管的数值如何显示，7-bit)

c)该模块是顶层模块，调用其他三个模块，形成一个自动售货机的系统：

```
module VEM(
    input clk, reset,
    input [2:0]No,
    input [3:0]money,
    output DP,DoneLED,
    output [7:0]AN,
    output [6:0]A2G
    );

    wire [5:0]money_paid;
    wire hasDone;

    paid calPaid(clk, hasDone, reset,
    No,
    money, money_paid);
    display display(clk, hasDone, No,
        money_paid, DoneLED, DP, AN, A2G);
    compDone compDone(No, money, money_paid, hasDone);
endmodule
```

三、仿真波形

1.module paid：

a)测试用例：根据需求分析，在此次实验中，测试了一下 7 中情况：

①商品编号无效，但是 money 有效→paid = 0

②商品编号为 4，money = 1 → paid = 1

③商品编号从 4 变为 2，没有 money 输入 → paid 保持 1

④保持商品编号为 2 不变，money = 1 → paid = 2，之后 hasDone 从 0 变为 1

⑤在④已经完成一次交易的基础上，保持商品编号不变，输入无效的 money，paid 不会清零

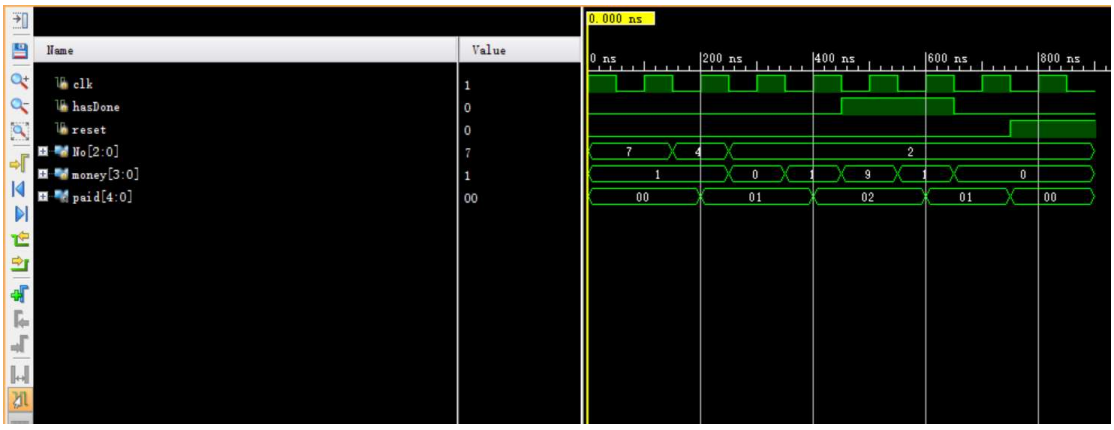
⑥保持商品编号不变，money = 1 → paid 清零之后变为 1

⑦测试 reset，将 paid 手动清零 → paid = 0

b)测试代码：

```
initial begin
        clk = 1;
        reset = 0;
        hasDone = 0;
        No = 3'b111;
        money = 4'b0001;
        #150 No = 3'b100; //150
        #100 money = 0; //250
        No = 3'b010; //250
        #100  money = 4'b0001; //350
        #100 hasDone = 1; //450
        money = 4'b1001; //450
        #100 money = 4'b0001; //550
        #100 hasDone = 0; money = 0; // 650
        #100 reset = 1; // 750
    end
```

c)仿真波形：



2.module compDone：

a)**测试用例：**该模块实现的功能是计算 hasDone，判断当前交易是否完成，因此对于商品编号为 4 的情况，price=2 测了一下四种情况：

①已输入金额为 5 角 → hasDone = 0

②已输入金额为 2 元 → hasDone = 1

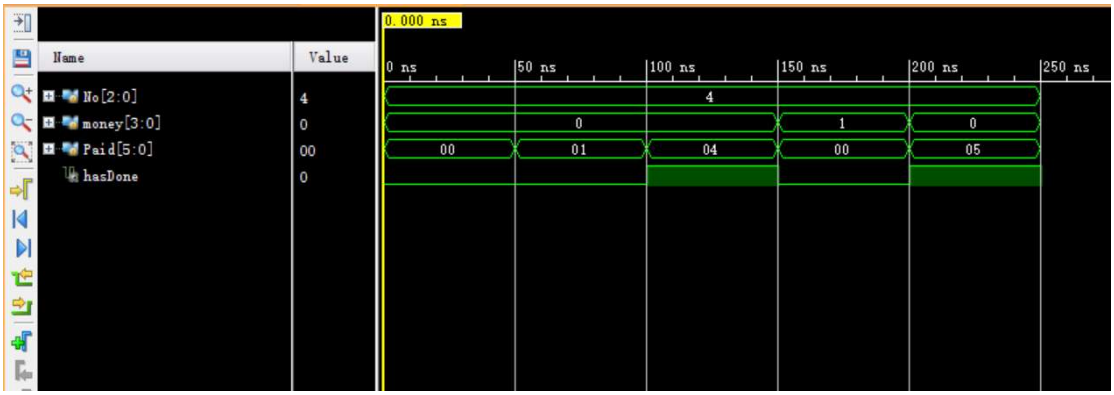
③在②交易已完成的情况下，测试输入新的 money，hasDone 的变化 → hasDone = 0

④超额支付，money = 0， 已输入金额为 2.5 元 → hasDone = 1

b)**测试代码：**

```
initial begin
    No = 3'b100;
    money = 4'b0000;
    Paid = 6'b000000;

    #50 Paid = 6'b000001;

    #50 Paid = 6'b000100;

    #50 Paid = 6'b000000;
    money = 4'b0001;

    #50 money = 4'b0000;
    Paid = 6'b000101;
end
```

c)仿真波形：



3.module display：

a)**测试用例：**该模块实现的是显示商品编号，已输入金额和交易完成状态，因此测了一下两种状态：

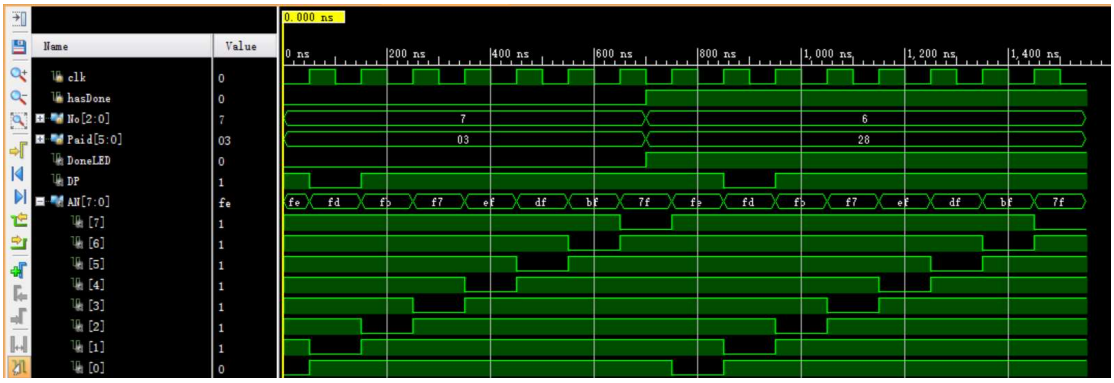
①商品编号为 7，已输入金额为 6’b000011，hasDone=0

②商品编号为 6，已输入金额为 6’b101000，hasDone=1

b)**测试代码：**

```
initial begin
    clk = 0;
    No = 3'b111;
    Paid = 6'b000011;
    hasDone = 0;

    #700 No = 3'b110;
    Paid = 6'b101000;
    hasDone = 1;
end
```

c)仿真波形：



4.module VEM：

a)**测试用例**：该测试模拟了一个真实的可能场景，流程如下：

①商品编号选定为 7，money 为 5 角

②保持 money 为 5 角，商品编号由 7 变为 4，money 变为 1 元

③商品编号由 4 变为 2，此时交易完成

④在交易完成的基础上，再投入无效的 money=4’b0101

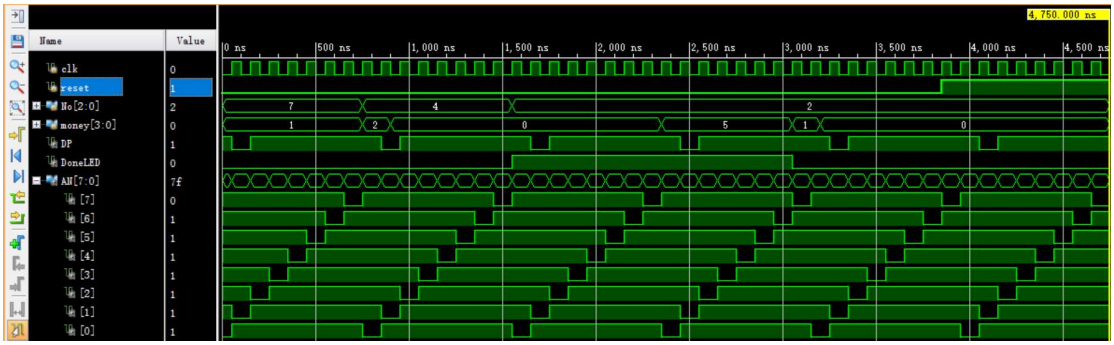
⑤投入有效的 money 为 5 角

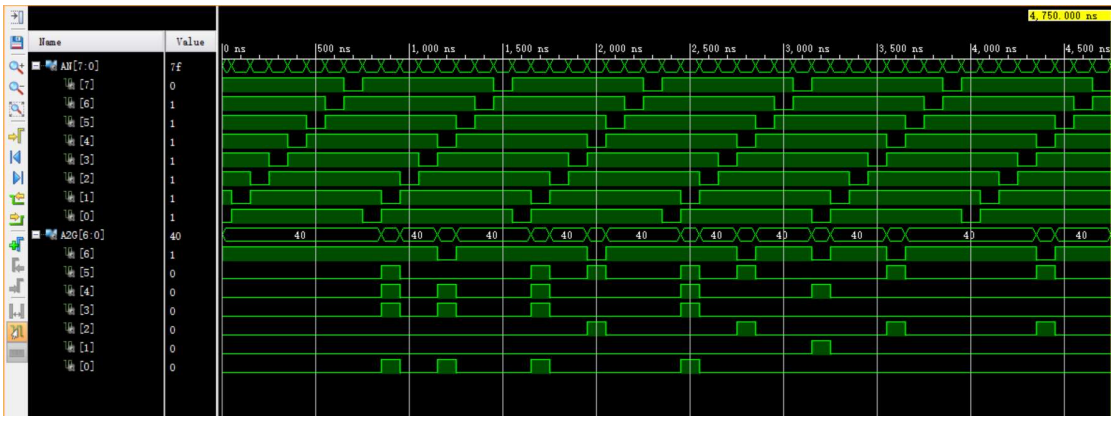
⑥reset

b)**测试代码**：

```
initial begin
        clk = 0;
        reset = 0;
        No = 3'b111;
        money = 4'b0001;
        #750 No = 3'b100;
        money = 4'b0010;
        #150 money = 0;
        #650 No = 3'b010;
        #800 money = 4'b0101;
        #700 money = 4'b0001;
        #150 money = 0;
        #650 reset = 1;
    end
```

c)仿真波形：





四、实验中遇到的问题及解决思路

1. 本次实验中，最重要的一个问题就是需求分析，因为不像之前的三次 LAB，文档中直接体现了整体的电路图关系和总输入、总输出，其中的很多细节在文档中也没有明确给出，因此我参考了老师的意见，并且联系实际生活，在实验中给出我的处理

2. hasDone 的初始值问题，刚开始我并没有对 hasDone 赋初始值，下板测试的时候，是正确的，但是，运行测试代码，查看仿真波形，发现 hasDone 在刚开一段时间内都是不确定状态，导致后面也出现了一小段异常，因此在 compDone 中添加了对 hasDone 赋初始值的代码

3.金额的小数部分显示问题，因为此次输入金额可能会包含一个 0.5 的小数部分，因此考虑到是定点小数，我对他们做了统一的处理，无论是 money_cur 还是 Paid，最后一位的 1 都表示是 5 角

五、实验中的收获与感受

整体感受是本次 LAB 较为简单，大部分功能都是之前 LAB 中已经实现的，比如：分频处理与上次 Q1 的显示中的分频处理原理相同，七段数码管的显示也已经在上次 LAB 中实践过了，此次 LAB 只不过是按照新的处理逻辑做了综合，并不复杂。