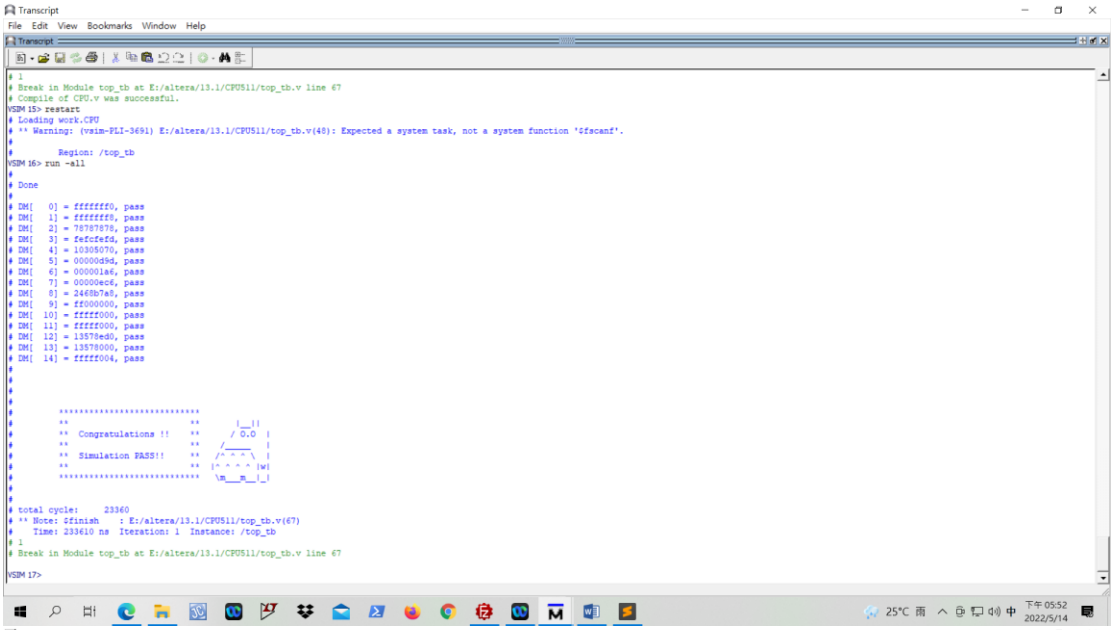
Computer Organization 2022

HOMEWORK 4

系級：資訊工程系 113　　學號：F74109016　　姓名：葉惟欣

實驗結果圖：

（波形圖及模擬完成截圖）



程式運作流程:

(簡單說明波形變化的意義)

```
module CPU(
    input             clk,
    input             rst,
    input      [31:0] data_out,
    input      [31:0] instr_out,
    output            instr_read,
    output            data_read,
    output reg [31:0] instr_addr,
    output reg [31:0] data_addr,
    output reg [3:0]  data_write,
    output reg [31:0] data_in
);

reg [2:0] CurrentState;
reg [2:0] NextState;
reg [31:0] Register[31:0];
reg [31:0] Immediate;

reg [31:0] old_instr_addr;
reg Instruction_Fetch;
reg Instruction_Decode;
reg Execute;
reg Memory_Access;
reg Write_Back;

integer i;

parameter   Idle_state = 3'h0,
            Instruction_Fetch_state = 3'h1,
            Instruction_Decode_state = 3'h2,
            Execute_state = 3'h3,
            Memory_Access_state = 3'h4,
            Write_Back_state = 3'h5,
            Finish_state = 3'h6;

wire [6:0]  opcode;
wire [4:0]  rd;
wire [2:0]  funct3;
wire [4:0]  rs1;
wire [4:0]  rs2;
wire [4:0]  shamt;
wire [6:0]  funct7;

assign instr_read = 1;
assign data_read = 1;

assign opcode = instr_out[ 6: 0];
assign rd =     instr_out[11: 7];
assign funct3 = instr_out[14:12];
assign rs1 =    instr_out[19:15];
assign rs2 =    instr_out[24:20];
assign shamt =  instr_out[24:20];
assign funct7 = instr_out[31:25];
```

用一個 old_instr_addr 來先前的 address，因為之後再 jalr 與 jar 指令時會 data dependency 有 Read after write 的問題。故再 write 前會先將原本的值存下來，這樣才會存到正確的值。

Immediate 的部分:

```
54  ////////////////////////////////////////////
55  //Immediate
56  always@(posedge clk or posedge rst)begin
57      if(rst)
58          Immediate <= 32'h0;
59      else if(Instruction_Decode)begin
60          case(opcode)
61              7'b0000011:begin//LW
62                  /*add your code*/
63                  Immediate[31:12] = {20{instr_out[31]}}; //sign
64                  Immediate[11:0] = instr_out[31:20];
65              end
66              7'b0010011:begin//I-type
67                  /*add your code*/
68                  Immediate[31:12] = {20{instr_out[31]}}; //sign
69                  Immediate[11:0] = instr_out[31:20];
70              end
71              7'b1100111:begin//JALR-type
72                  /*add your code*/
73                  Immediate[31:12] = {20{instr_out[31]}}; //sign
74                  Immediate[11:0] = instr_out[31:20];
75              end
76              7'b0100011:begin//S-type
77                  /*add your code*/
78                  Immediate[31:12] = {20{instr_out[31]}}; //sign
79                  Immediate[11:5] = instr_out[31:25];
80                  Immediate[4:0] = instr_out[11:7];
81              end
82              7'b1100011:begin//B-type
83                  /*add your code*/
84                  Immediate[31:12] = {20{instr_out[31]}};
85                  Immediate[11] = instr_out[7];
86                  Immediate[10:5] = instr_out[30:25];
87                  Immediate[4:1] = instr_out[11:8];
88                  Immediate[0] = 1'b0;
89              end
90              7'b0010111:begin//AUIPC
91                  /*add your code*/
92                  Immediate[31:12] = instr_out[31:12];
93                  Immediate[11:0] = 12'h0;
94              end
95              7'b0110111:begin//LUI
96                  Immediate[31:12] = instr_out[31:12];
97                  Immediate[11:0] = 12'h0;
98              end
99              7'b1101111:begin//J-type
100                 /*add your code*/
101                 Immediate[31:20] = {12{instr_out[31]}};
102                 Immediate[10:1] = instr_out[30:21];
103                 Immediate[11] = instr_out[20];
104                 Immediate[19:12] = instr_out[19:12];
105                 Immediate[0] = 1'b0; //J-immediate encodes a signed offset in multiples of 2 bytes.
106             end
107         endcase
108     end
109 end
110 ////////////////////////////////////////////////
111
```

The load are encoded in the I-type, indirect jump instruction JALR (jump and link register) 用跟 I-mmediate 一樣的方式。

The 12 bit B immediate encodes signed offsets in multiples of 2(因此最後一個 bit 設為 0。

AUIPC (add upper immediate to pc) is used to build pc-relative address and uses the U-type format, AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros.(所以跟 LUI immediate 方式相同)

Immediate 如何從 instr_out 擷取出來，我直接參考 Spefication 的內容，裡面有提到哪些 type 需要 sign extension: (I-immediate,S-immediate,B-immediate,J-immediate)，而 U-immediate 則不用。

J-type 含的指令為(jal) 的 immediate encodes a signed offset in multiples of 2 bytes(也因此最後一個為 0)。The offset is sign extended and added to pc 再存到 rd。

```
//////////////////////////////////////////////
//Register_Files
always@(posedge clk or posedge rst)begin
    if(rst)begin
        for(i = 0; i < 32; i = i + 1)
            Register[i] <= 32'h0;
    end
    else if(Write_Back)begin
        case(opcode)
            7'b0110011:begin//R-type
                case(funct3)
                    3'b000:begin
                        case(funct7)
                            7'b0000000://ADD
                                Register[rd] = Register[rs1] + Register[rs2];
                            7'b0100000:begin//SUB
                                /*add your code*/
                                Register[rd] = Register[rs1] - Register[rs2];
                            end
                        endcase
                    end
                    3'b001:begin
                        case(funct7)
                            7'b0000000://SLL
                                Register[rd] = Register[rs1] << Register[rs2][4:0];
                        endcase
                    end
                    3'b100:begin
                        case(funct7)
                            7'b0000000:begin//XOR
                                /*add your code*/
                                Register[rd] = Register[rs1] ^ Register[rs2];
                            end
                        endcase
                    end
                    3'b110:begin
                        case(funct7)
                            7'b0000000:begin//OR
                                /*add your code*/
                                Register[rd] = Register[rs1] | Register[rs2];
                            end
                        endcase
                    end
                    3'b111:begin
                        case(funct7)
                            7'b0000000:begin//AND
                                /*add your code*/
                                Register[rd] = Register[rs1] & Register[rs2];
                            end
                        endcase
                    end
                endcase
            end
            7'b0000011:begin
                case(funct3)
                    3'b010:begin//LW
                        Register[rd] = data_out;
                    end
                endcase
            end
            7'b0010011:begin//I-type
                case(funct3)
                    3'b000:begin//ADDI
                        Register[rd] = Register[rs1] + Immediate;
                    end
                    3'b100:begin//XORI
                        /*add your code*/
                        Register[rd] = Register[rs1] ^ Immediate;
                    end
                    3'b110:begin//ORI
                        /*add your code*/
                        Register[rd] = Register[rs1] | Immediate;
                    end
                    3'b111:begin//ANDI
                        /*add your code*/
                        Register[rd] = Register[rs1] & Immediate;
                    end
                endcase
            end
            7'b1100111:begin//JALR
                case(funct3)
                    3'b000:begin
                        /*add your code*/
                        old_instr_addr = instr_addr + 32'd4;
                    end
                endcase
            end
        7'b0010111:begin//AUIPC
          /*add your code*/
          Register[rd] = instr_addr + Immediate;
        end
        7'b0110111:begin//LUI
          /*add your code*/
          Register[rd] = Immediate;
        end
        7'b1101111:begin//J-type
          /*add your code*/
          if(rd != 32'h0)
          begin
            Register[rd] = instr_addr + 32'd4;
          end
        end
      endcase
    end
end
//////////////////////////////////////////////
```

Jalr : jump and link register。
會跳到新的 instr_addr，但當函數返回(return 時)，會回到現在 instr_addr 的下個位址也就是 instr_addr+4，故要將返回的位址存起來。原本應該是將返回的位址寫在 Register[rd]裡面，但因為 target instr_addr 由 Immediate 與 Register[rs1]相加，如果 rs1 與 rd 剛好相同會導致 Register[rd]在 jump 後 instr_addr+4 才被更改到原先的 Register[rs1]的值，改變了值得 Register[rs1]並不是原先 instr_addr = Immediate + Register[rs1]所要的值應此這兩個動作應該先用不同的暫存器來存取，有 WAR 的 data dependency。

AUIPC 指令: 將 immediate 加到 PC 中，然後存到 register rd 中，為什麼 AUIPC 不會有 jalr 的問呢。因為 AUIPC 通常搭配 jalr，因為 jalr 為(12 位 immediate)加上 instr_addr 而得到 target address. 二 auipc 則是有 32bit 20 bit – Uimmediate，filling the lowest 12 bits with zeros.這兩個指令一起用 instr_addr 就可以到 32bit PC relative address 的任何位址了。

J type 裡的指令 : jal (Jump and link) stores the address of the instruction following the jump (pc +4) into the register rd.但 instr_addr 不會用到 register file 因此不會造成如 jalr 的問題，唯一要注意的事，Register[rd] 有可能未暫存器 0，所以要避免掉，●後面有詳細說明。

```
218  //instr_addr == PC
219  always@(posedge clk or posedge rst)begin
220      if(rst)
221          begin
222              instr_addr = 0;
223          end
224      else if(Write_Back)begin
225          case(opcode)
226              7'b1100111:begin
227                  case(funct3)
228                      3'b000:begin//JALR
229                          /*add your code*/
230                          instr_addr = (Register[rs1] + Immediate) & {{31{1'b1}},1'b0};
231                          if(rd != 32'h0)
232                          begin
233                              Register[rd] = old_instr_addr;
234                          end
235                      end
236                  endcase
237              end
238              7'b1100011:begin//B-type
239                  case(funct3)
240                      3'b000:begin//BEQ
241                          /*add your code*/
242                          if(Register[rs1] == Register[rs2])
243                              instr_addr = instr_addr + Immediate;
244                          else
245                              instr_addr = instr_addr + 32'd4;
246                      end
247                      3'b001:begin//BNE
248                          /*add your code*/
249                          if(Register[rs1] != Register[rs2])
250                              instr_addr = instr_addr + Immediate;
251                          else
252                              instr_addr = instr_addr + 32'd4;
253                      end
254                      3'b111:begin
255                          if(Register[rs1] >= Register[rs2])
256                              instr_addr = instr_addr + Immediate;
257                          else
258                              instr_addr = instr_addr + 4;
259                      end
260                  endcase
261              end
264          7'b1101111:begin//JAL-type
265            /*add your code*/
266            instr_addr = instr_addr + Immediate;
267
268          end
269          default:begin//default
270            /*add your code*/
271            instr_addr = instr_addr + 32'd4;
272          end
273      endcase
274    end
275  end
276  ////////////////////////////////////////////////
```

Jalr 的指令:
Target address 是由 12 bit signed Immediate 加上 register rs1，然後 set the least significant bit of the result to zero 來的，也就是第 230 行。而第 231 行到第 234 行是如同上面講的如果直接在 register file 中寫 Register[rd] = instr_addr + 32'd4，會有將 230 行的 instr_addr 用到剛剛被改值的 Register[rs1]的問題，因此這裡先是做完 230 行後才將 Register[rd] = Register[rs1]改值，也因此寫法用 blocking。如果不這樣寫 jalr 的指令會出錯。

判斷 rd 是否為 0，因為在一開始 set up 時，如果沒有避免修改到 register[0]的值 register[0]的值會被修改而不再是 0，如此一來會造成之後 addi,sub,xor,and,addi,xori,ori 的指令結果出問題。而 **jal** 也是同樣的問題。

B-type 的 instr_addr 就直接為 12-bit B immediate encodes signed offsets in multiple of 2, and is added to the current pc to give the target address.

Jal: The offset is sign – extended and added to the pc to form the jump taget address.s

```
280   //data_addr
281   always@(posedge clk or posedge rst)begin
282       if(rst)
283           data_addr <= 32'h0;
284       else if(Execute)begin
285           case(opcode)
286               7'b0000011:begin//L-type
287                   /*add your code*/
288                   data_addr <= Register[rs1] + Immediate;
289               end
290               7'b0100011:begin//S-type
291                   /*add your code*/
292                   data_addr <= Register[rs1] + Immediate;
293               end
294           endcase
295       end
296   end
297   ////////////////////////////////////////////////
298
299   ////////////////////////////////////////////////
300   //data_write
301   always@(posedge clk or posedge rst)begin
302       if(rst)
303           data_write <= 4'h0;
304       else if(Execute)begin
305           case(opcode)
306               7'b0100011:begin
307                   case(funct3)
308                       3'b010:begin//SW
309                           data_write <= 4'hf;
310                       end
311                   endcase
312               end
313           endcase
314       end
315       else if(Memory_Access)
316           data_write <= 4'h0;
317   end
318   ////////////////////////////////////////////////
319
320   ////////////////////////////////////////////////
321   //data_in
322   always@(posedge clk or posedge rst)begin
323       if(rst)
324           data_in <= 0;
325       else if(Execute)begin
326           case(opcode)
327               7'b0100011:begin
328                   if(Register[rs1][1:0] + Immediate[1:0] == 2'b0)
329                       data_in <= Register[rs2];
330               end
331           endcase
332       end
333   end
334   ////////////////////////////////////////////////
335
```

L-type 與 S-type 一個是要讀 memeory，一個是要寫 memory，而要讀寫的資料分別放在 data_addr 由 Register[rs1] + immediate 得到。

特別說明:

RISC-V pseudoinstructions.　Ret　jalr x0,x1,0 Return from subroutine 改到 0 的暫存器。 Set up 的過程。

```
00000084 <init_bss>:
   84: 00008517          auipc  a0,0x8
   88: 07c50513          addi   a0,a0,124 # 8100 <__bss_end>
   8c: 00008597          auipc  a1,0x8
   90: 07058593          addi   a1,a1,112 # 80fc <_test_start+0xfc>
   94: 00000613          li     a2,0
   98: 06c000ef          jal    ra,104 <fill_block>
```

```
0000009c <init_sbss>:
   9c: 00008517          auipc  a0,0x8
   a0: 06450513          addi   a0,a0,100 # 8100 <__bss_end>
   a4: 00008597          auipc  a1,0x8
   a8: 05858593          addi   a1,a1,88 # 80fc <_test_start+0xfc>
   ac: 00000613          li     a2,0
   b0: 054000ef          jal    ra,104 <fill_block>
```

```
000000b4 <write_stack_pattern>:
   b4: 00008517          auipc  a0,0x8
   b8: 04c50513          addi   a0,a0,76 # 8100 <__bss_end>
   bc: 00009597          auipc  a1,0x9
   c0: 04058593          addi   a1,a1,64 # 90fc <_gp+0x7fc>
   c4: 00000613          li     a2,0
   c8: 03c000ef          jal    ra,104 <fill_block>
```

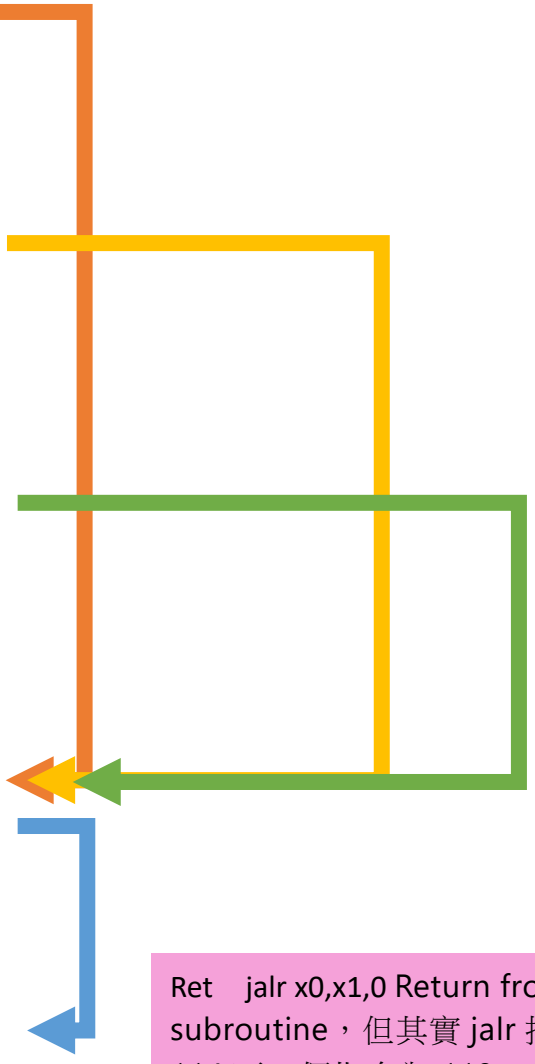
```
000000cc <init_stack>:
   cc: 00009117          auipc  sp,0x9
   d0: 03410113          addi   sp,sp,52 # 9100 <__stack>
```

```
000000ec <SystemInit>:
   ec: 02c000ef          jal    ra,118 <main>
```

```
00000104 <fill_block>:
  104: 00b57863          bgeu   a0,a1,114 <fb_end>
  108: 00c52023          sw     a2,0(a0)
  10c: 00450513          addi   a0,a0,4
  110: ff5ff06f          j      104 <fill_block>
```

```
00000114 <fb_end>:
  114: 00008067          ret
```

```
00000118 <main>:
  118: ffc10113          addi   sp,sp,-4
  11c: 00812023          sw     s0,0(sp)
  120: 00008417          auipc  s0,0x8
  124: ee040413          addi   s0,s0,-288 # 8000 <_test_start>
```



Ret　jalr x0,x1,0 Return from subroutine，但其實 jalr 指令位址為 114 下一個指令為 118，並不會再繼續執行下一個指令，因此其實不需要存下一個指令的位址 118，而他又是存在 register 0 的位址，不存的好處就是 register 0 位址不會被覆蓋掉。

Jalr 與 jal，Jalr 的指令 description 為: jalr rd rs1 signoff(Immediate).

Instr_addr = Register[rs1] + signoff;

Register[rd] = instr_addr + 4;

3b4: 00030367　　0000 0000 0000 0011 0000 0011 0110 0111

Opcode 1100111

rd = 00110

funct3 = 000

assign rs1 =　 0011 0

immediate = 0

jalr　t1,t1

1. Register[rd]= instr_addr + 4;
2. Instr_addr = Register[rs1] + immediate (0);

此時 instr_addr 會變成 instr_addr + 4 + immediate (0) = instr_addr + 4;
但要得其實是原先的 register[t1] 也就是還未被第一個指令 Register[t1] =instr_addr + 4 前的 Register[t1]的值，因為指令 30367 的 rd 與 rs1 相同所以會造成這個問題，所以才要先把原先要放到 Register[rd]的值先放在其他地方，等第二個指令結束再執行真正的第一行指令。

心得

（請寫下完成本次作業的心得、學到哪些東西、困難點的部分。）

這次的作業又學習到許多指令，像是 AUIPC 通常都是要跟 jalr 一起使用，如此一來可以到就可以到 32bit PC relative address 的任何位址。此外我還看了 The RISC-V Instruction Set Manual 裡面有很多詳細的說明，像是 conditional branch range is ± 4KiB. Jal can jump in range ± 1KiB. 其中最令我驚訝的 jalr 裡面的最後一句話，Register x0 can be used as the destination if the result is not required.雖然不確定我的寫法到底是不是正確的，但如同我在程式流程說明的部分提到 setup 過程中 ret 指令(jalr x0, x1, 0 Return from subroutine)，他接下來的 PC+4 addree 的確不需要再用到。這讓我覺得這個作業跟官方講的相互呼應，透過一直 debug 讓我對這種硬知識較為理解，而不只是看看書，之後就會忘記了。

在 debug 中，這次與以往不一樣的是，以前都是看波形圖，這次我是透過 display 的方式去對一個一個指令，也漸漸看懂到底助教出的是哪些指令，我要怎樣分析出示哪到指令出錯了。也因此我看懂 setup.S 與 main.S 還有 main.log 大致上在尬麻，像是裡面常用到的 li，我也去查了一下，li -- load immediate li rd,constant.跟 ret 依樣都是叫 pseudoinstruction 都可由其他指令轉過來，但我查了 Specification，卻發現，li rd,immediate 的 base instruction 寫 myriad sequences，現在還不太理解，可能之後有空再看，而 ret return from subroutine 的 base instruction 為 jalr x0,x1,0。在 main.log 中，每個指令的位址還有指令究竟為何都寫得很清楚，我透過 display 將覺得有問題的指令位址區段在

runtime 中顯示他的 instr_out，還有 rd 與 immediate 的一些資訊來慢慢 debug。起初，我就有發現這次指令錯掉的部分都是上次作業對的，而上次基本上都是基礎的 addi，sub, xor, and, add, xori, ori 的指令，但上次對了所以基本上 immediate 沒問題，因為實在找不出原因我便試圖去理解到底 fffffff0 與 fffffff8, 78787878, 10305090, 00000d9d, 000001a6, 00000ec62 到底是甚麼東西，自此我也發現更好玩的東西都在 main.log 裡面。

之後我理解原來指令的側資在這裡，因此我就慢慢去對，還跟作業四的 main log 做比較，比較才發現原來 main 以前的 setup 過程有修改過，既然如此問題可能就出在 setup 過程了，再慢慢一步步發現原來是 register0 的值會被修改到。找到是 register 0 的問題後，我就又開始思考甚麼東西會修改到 register 0，我便想說應該就可能是 jalr 與 jal，因為我可能不小心 jump 到錯誤的指令，然後不小心改到他了(雖然最後發現不是，是指令原本就寫在那裏)，因此就開始從這兩個指令著手，也因為從這兩個指令著手，我才又發現 jalr 有 rd 與 rs1 相同的問題，再去修改我的程式碼，但改了(像是多一個暫存器變數與 blocking)。

可惜雖然有解決一個問題但 register 0 為何被修改到始終沒找到，因此我只好去跑 setup 每個指令了，也是因為這個過程讓我對如何 setup 有一點頭緒，最後也發現 ret 這個神奇的指令。

做完這次作業，印證了發現問題比解決問題還重要，起初我以為又是 immediate 的問題，雖然剛開始在 immediate 的確也花了不少時間處理，但 immediate 始終是小問題，只要了解 instr_out 如何對應的即可。但因為有先前的成功經驗，即便我 immediate 已經寫對了，我仍不停的花時間在他身上。最後我才發現還有其他問題，當一旦找到問題之後，解決其實已經相對容易了，因為至少已經有了方向下手，很感謝這次的作業讓我學到很多，非常有成就感。