Maze-solving FPGA system

組別: 你選的都隊 | 組員 : 111062118 江佩霖、113062221 劉廷暉

I. 設計概念/示意圖

目的：實現有兩種模式的自走車，能成功跟據路線行走至終點。

說明：

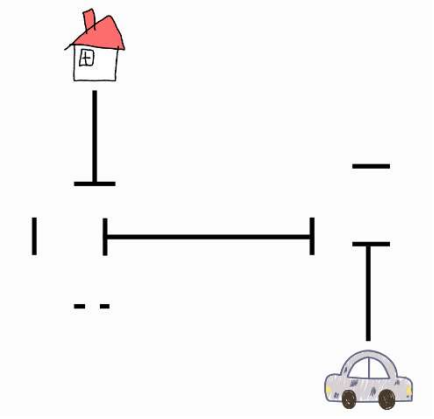
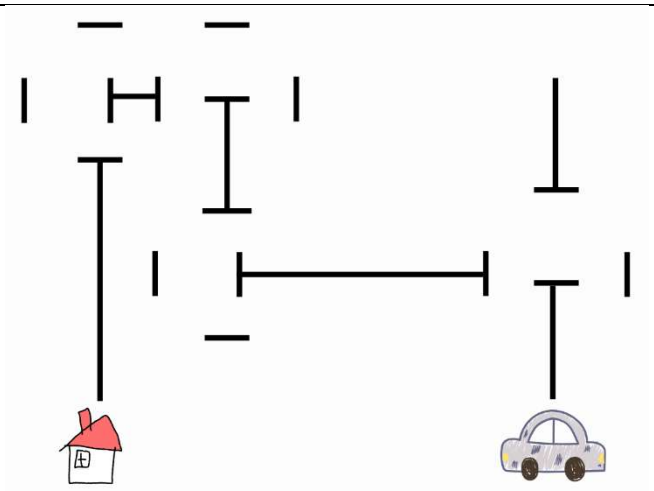
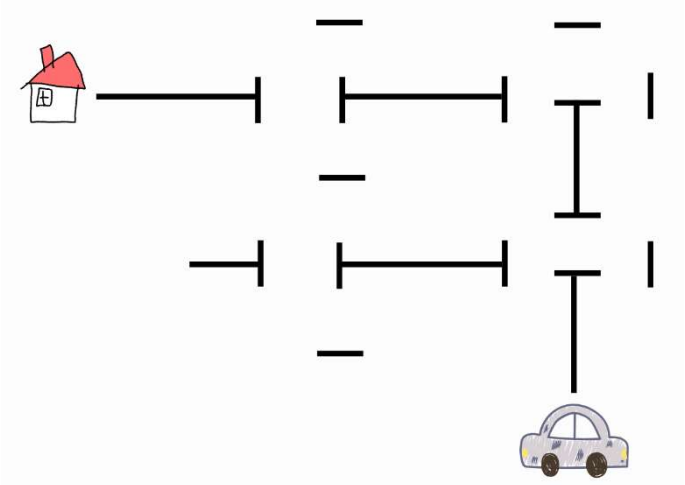
地圖 總共三張地圖，一張 Basic 以及兩張 Advance。

模式 車子有兩種模式 (開關控制)，可選擇讓車子自己偵測行走路線或是讓玩家控制車子在路口的行為。

控制 玩家能使用 FPGA 板上配置的搖桿來控制在轉彎路口時車子的行動，分別可以選擇上(直走)、左(左轉)以及向右行走。操作方式則是使用搖桿後按下搖桿的按鈕確認輸入，車子才會接收訊號行走。

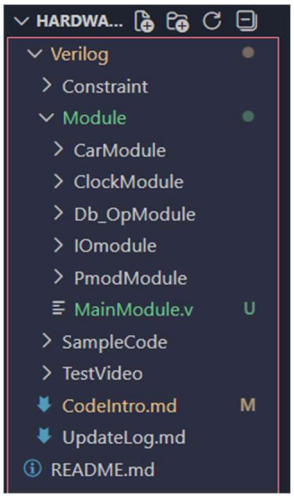
終點 車子會偵測前方障礙物，設計偵測到障礙物時判定抵達迷宮終點。

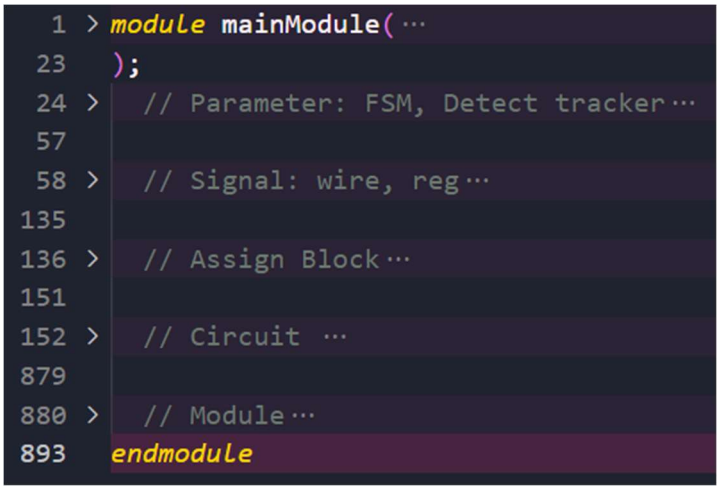
地圖：一共設計三張地圖，起點、終點以及路線如下表所示

Basic	Advance1 & 2
 NOTE: Basic 地圖僅作為測試使用，不支援玩家控制模式。	 

II. 架構細節及方塊圖

i. Code Structure & Reference





➢ Main Module Structure

Block	Include	Function
Parameter	FSM, Detect tracker, Mem's localparam	Define all parameter we need in the system.
Signal	System, Counter, IO siganl	Define all reg, wire we need in the system.
Assign	Distance, Counter enable, IO display, Pmod	Assign a value to the wire signal if needed.
Circuit	System Circuit & IO Circuit	Sequential Block and Combinational Block.
Module	Clk, Car, IO, Pmod…	Link the Submodule into Main Module

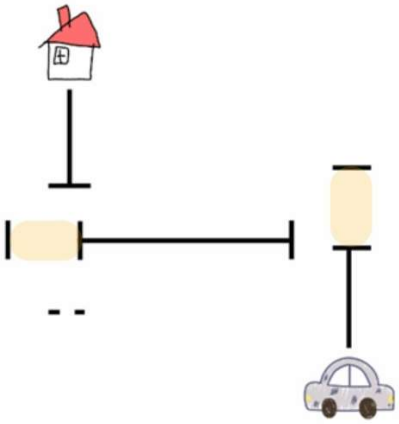
➢ GitHub repo (private, will turn to public if needed) :

https://github.com/ivylingchiang/NTHU_HardwareDesign-FinalProject.git

Module Dir	Include Modules	Function
Car Module	Motor Sonic Tracker	Control car detection and move. Ref. CS210401 Lab & Independent Work
Clock Module	System's Clock Modules	Process all the functions that we need to count. Ref. CS210401 Lab & Independent Work
Db Op Module	Debounce One Pulse	Processing signal. Ref. CS210401 Lab
IO Module	Seven Segment	Handling I/O output requests from the main module Ref. CS210401 Lab & Independent Work
Pmod Module	Pmod JSTK2	Process all the functions that we need to count. Ref. Here, Tutorial & Independent Work

- **地圖設計**:

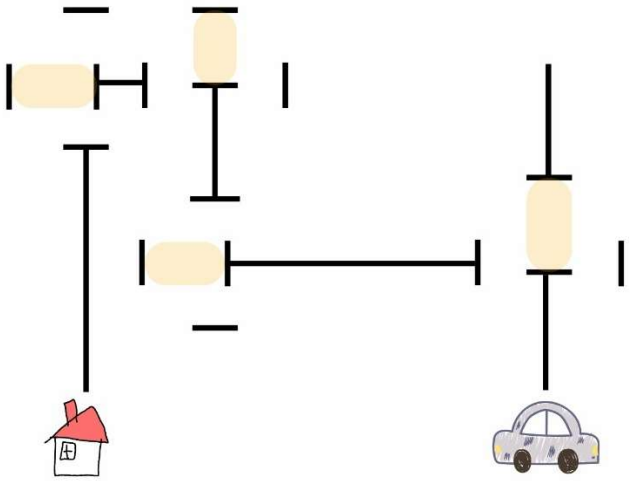
  在每個轉角都設置待轉區，由兩個全黑跑道形成的區域(如下圖黃色區塊)，車子會在偵測到第一個全黑跑道狀態後進入 CHOOSE state，並且在偵測到第二個全黑跑道時做出轉向決定(AUTO mode)或是等待使用者輸入(Manual mode)。



- **模式說明**

  自走車模式下，系統會依照直走、左轉、右轉的固定順序偵測當前路口各轉向是否可通行，並透過一個 stack 結構紀錄目前所選擇的行走路線；當車輛完成整個迷宮探索並抵達終點後，系統會將完整的行走路徑加以顯示。相對地，在使用者模式中，車輛於每個轉彎區域會停止並等待使用者透過搖桿輸入欲行進的轉向方向，並在按下按鈕確認後進行判斷；當系統確認該轉向方向為可通行路徑時，車輛即直行至下一個轉彎區域，並重複上述流程，直到成功抵達終點為止。

  以 Advance 1 地圖為例：



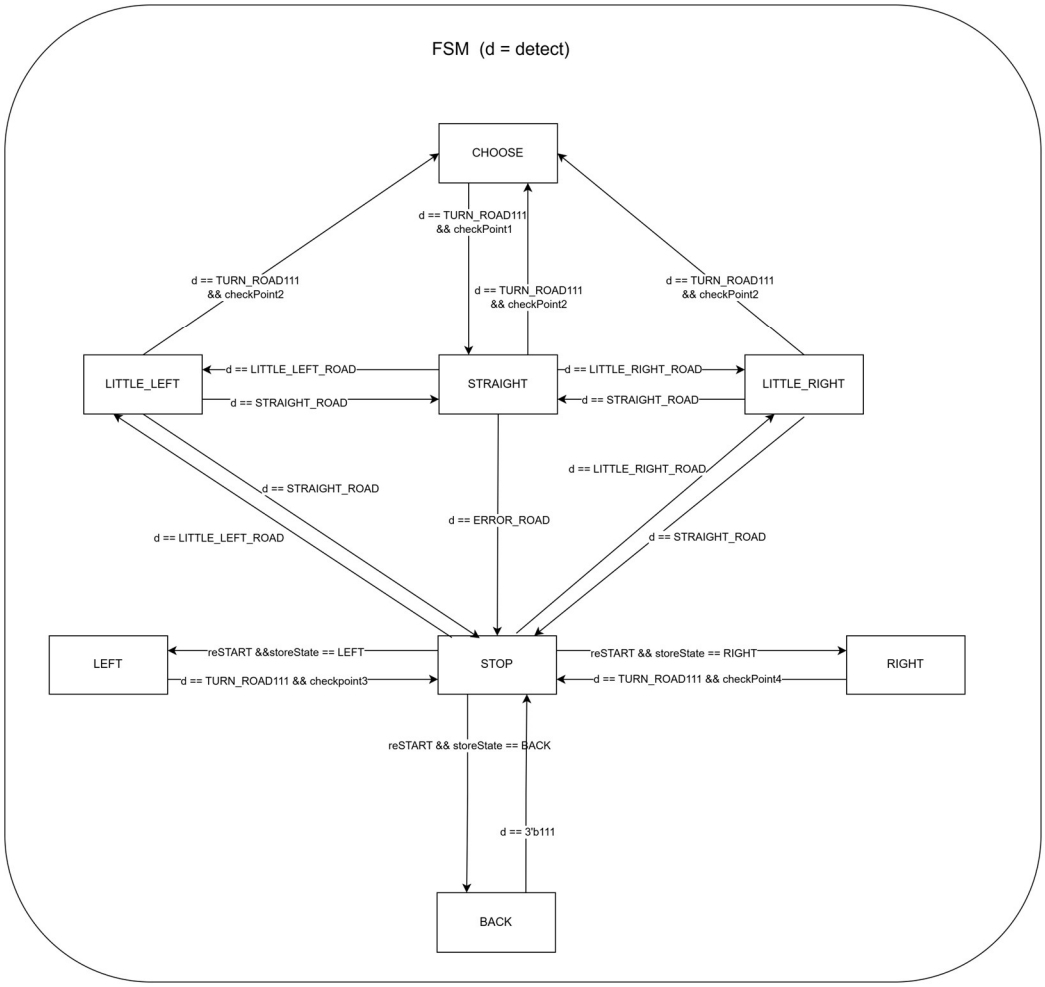
Stack 變化：

(第一個路口) 1
2
2 (第二個路口) 1
2 (第二個路口) 2
2 3
…
2 3 2 2 (終點)

迷宮路線：2 3 2 2

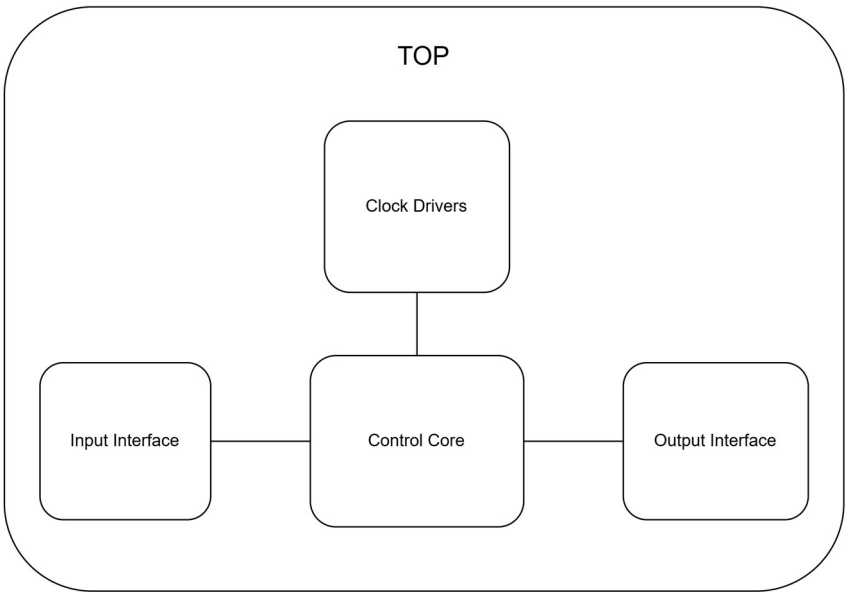
Note : 1(直走)；2(左轉)；3(右轉)

ii. FSM

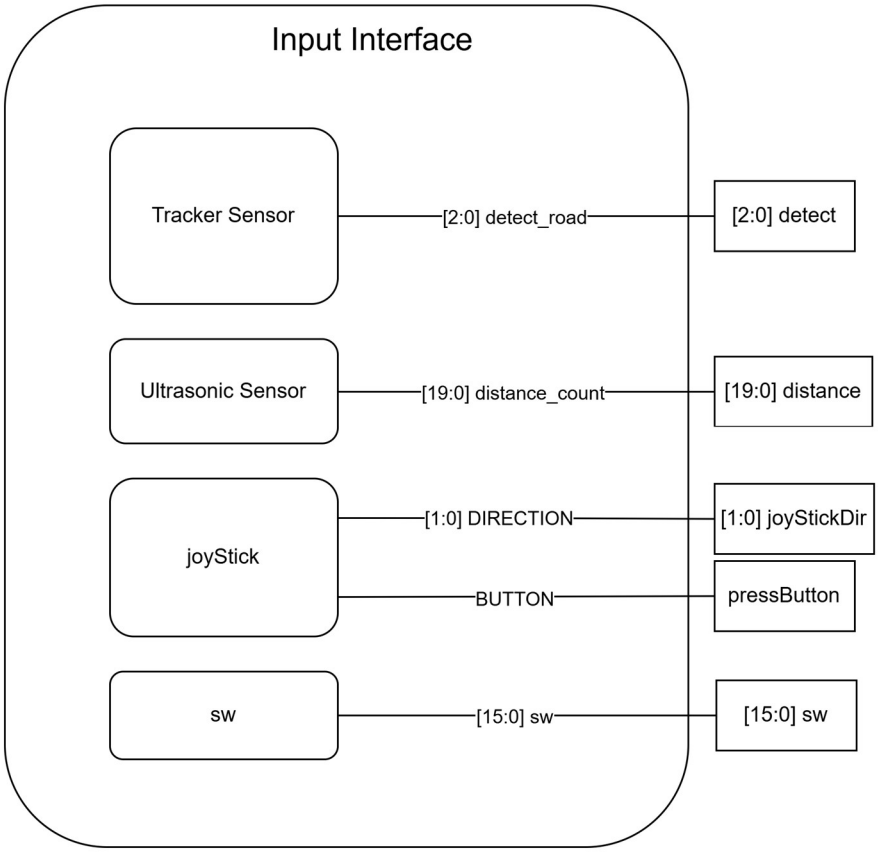


iii. Block Diagram

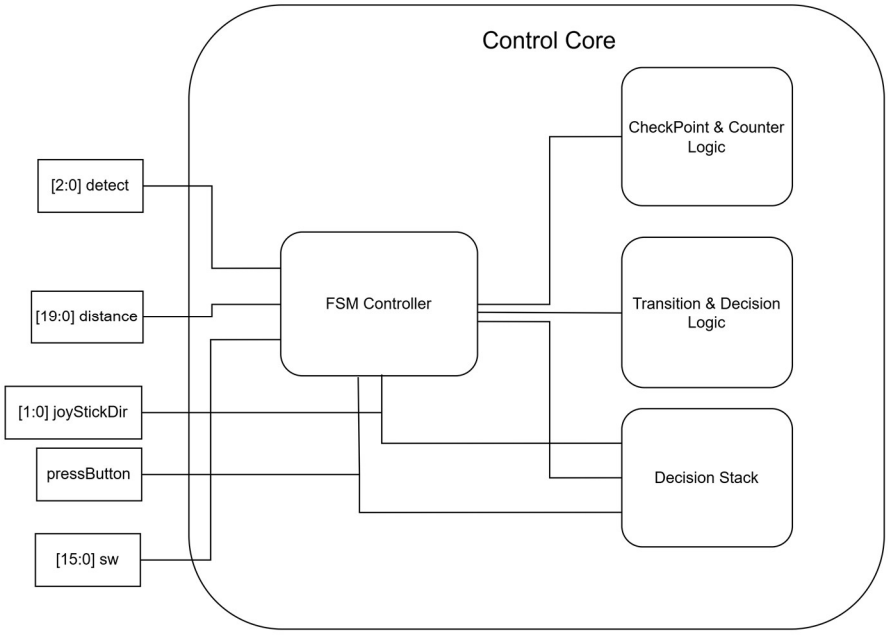
整體程式可大致分為以下四個部分：來自搖桿、超音波感測器、黑白感測器以及 FPGA 板開關的 Input Interface，控制馬達運作的 Control core，控制所有時間變化的 Clock Drivers 以及將數據輸出給馬達、七段顯示器和 LED 燈的 Output Interface。



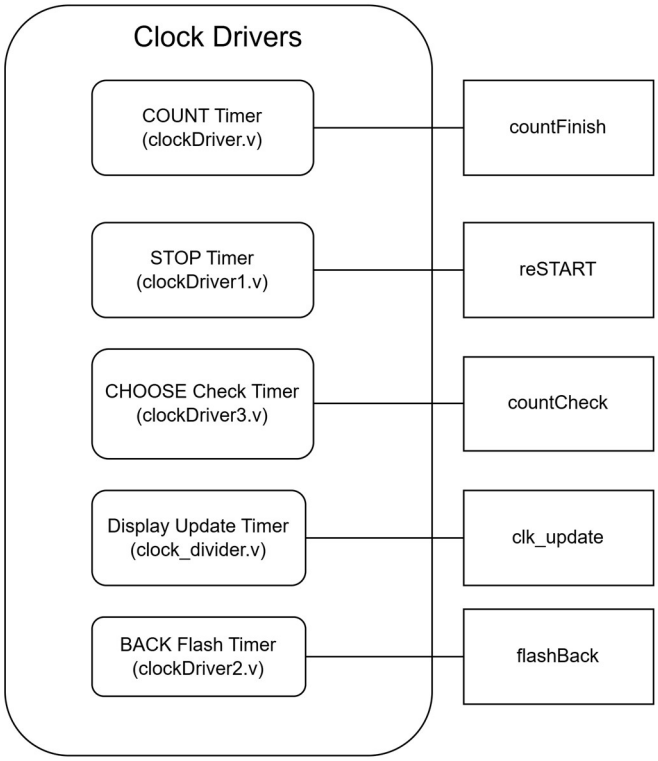
Input Interface 包含 3 bit 的 Tracker Sensor，負責將車前的黑白探測器數據回傳給 Control Core。Ultrasonic Sensor 負責在遇到障礙物時結束程式(即抵達終點)。joyStick 負責在手動模式下時，將玩家選擇的方向以及確認選擇時按下的決策傳給 Control Core。sw 負責切換手動與自動模式以及啟動車輛。



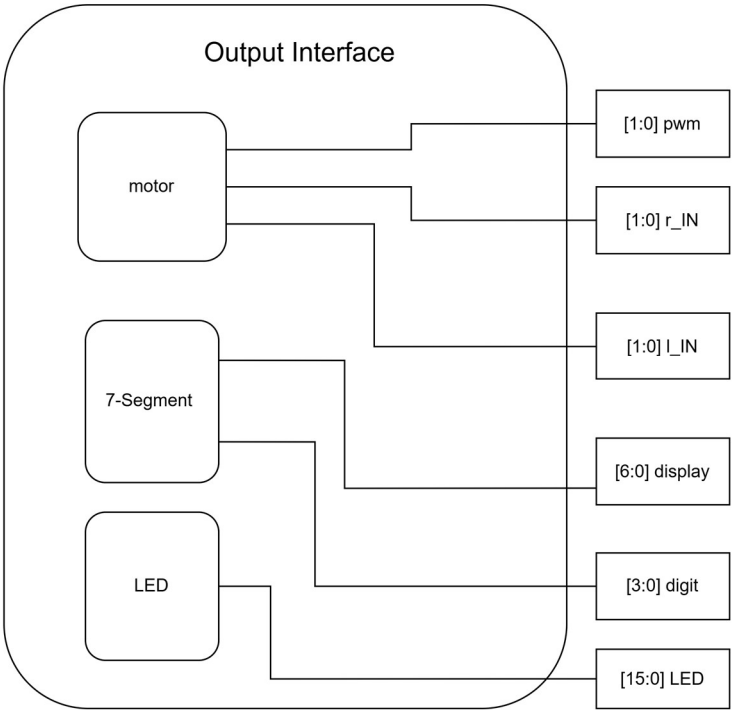
Control Core 由 FSM controller 為核心，根據 CheckPoint、Transition Logic 和 stack 的數據做 state transition。



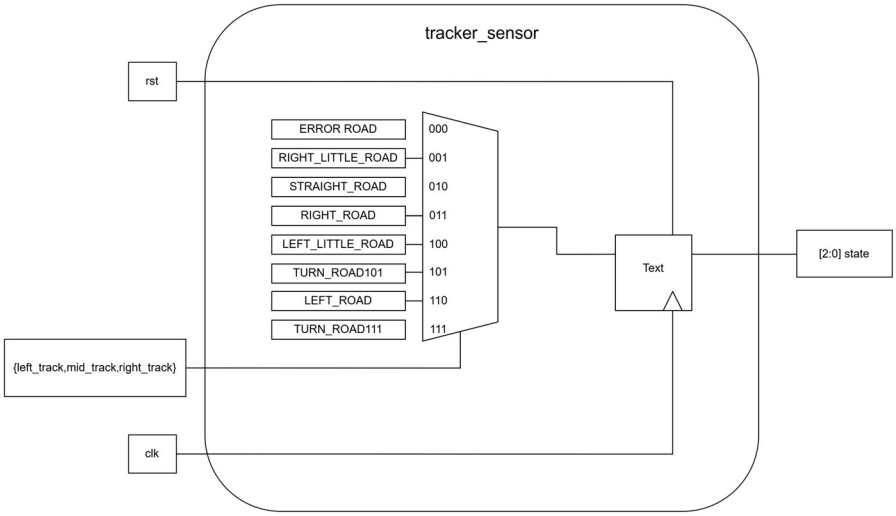
Clock Drivers 包含所有控制 clk 以及記數的功能，COUNT Timer 控制起跑倒數，STOP Timer 在車輛須停止並變更輪子轉向時，控制車子的停止秒數，CHOOSE Check Timer 在車輛手動模式下，使用者作出方向選擇後，等待 1 秒後車子才會接收並做出相應動作，BACK Flash Timer 則在自動模式下，於右轉時須連續偵測兩個路口時啟用。



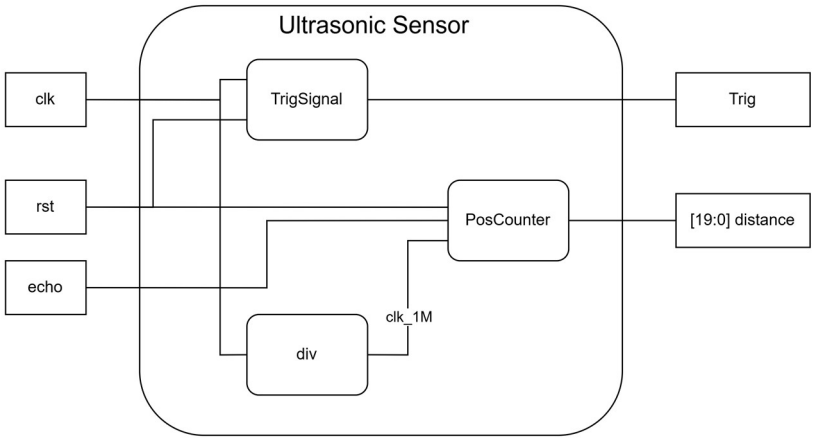
Output Interface 包含三個部分：接收 Control Core 數據並改變馬達轉速及轉向的 motor、顯示即時判斷情況的 7-segment 以及 LED。



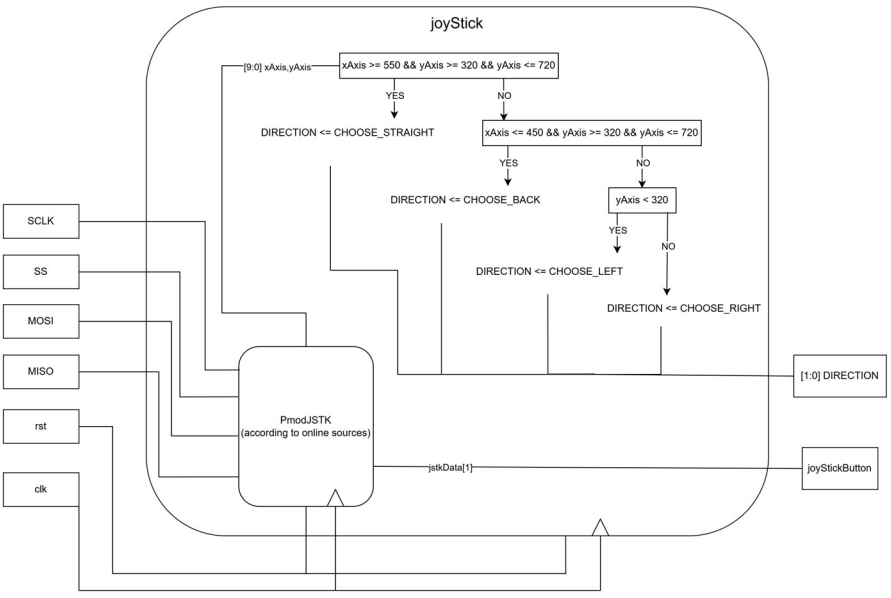
Tracker Sensor 根據黑白感測器的數據決定 state 並讓 Control Core 根據它做即時更新。



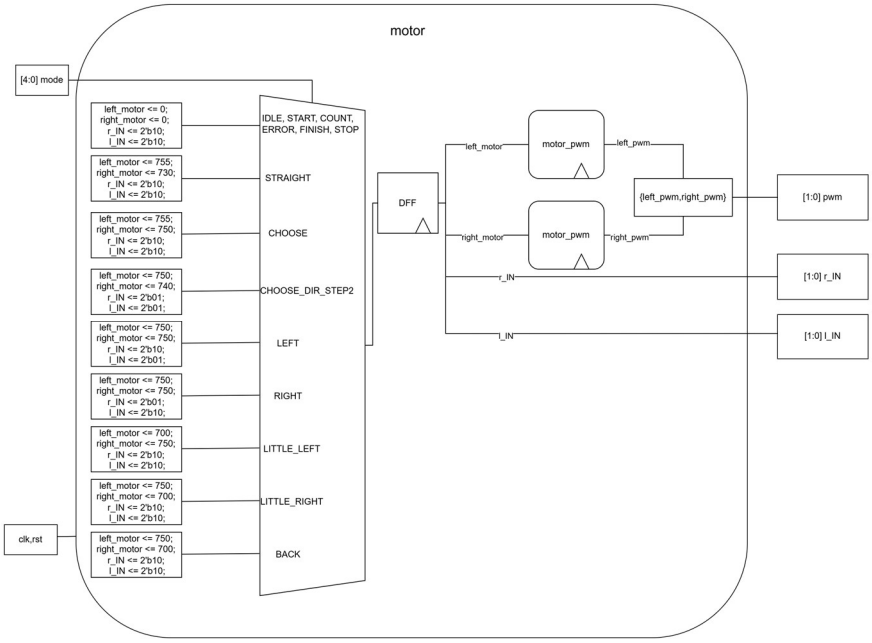
超音波感測器偵測到終點障礙物時，正確識別並停止車輛。



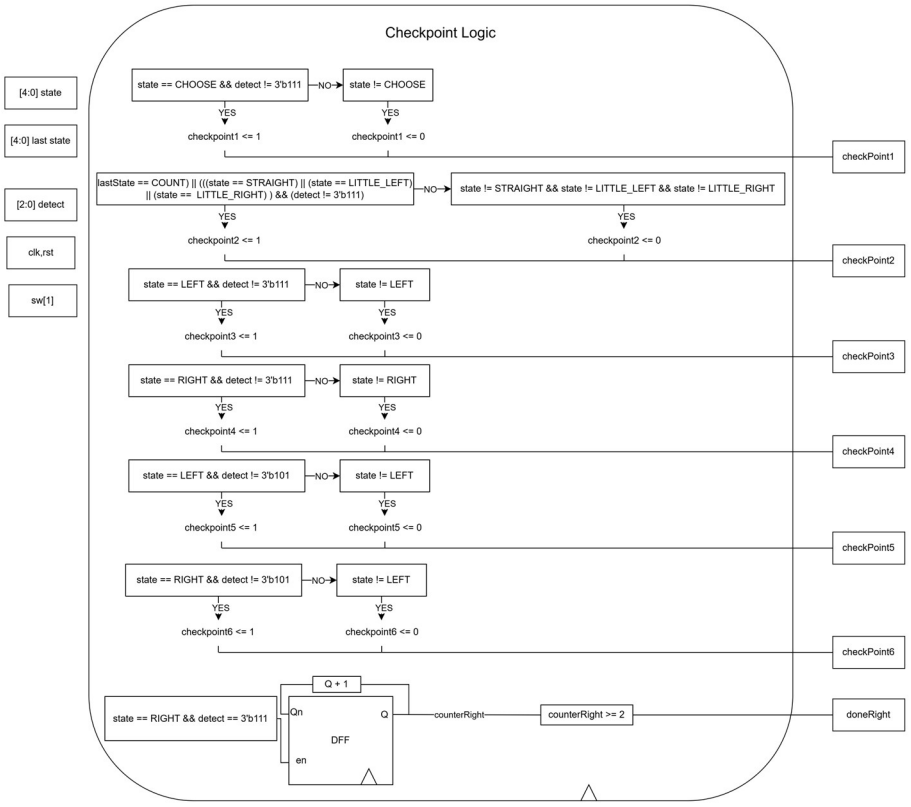
搖桿模組取自網路，經加工後把原先設計之輸出 X 及 Y 方向的位移量包裝成前後左右共四種結果並傳給 Control Core，按鈕設計則不做改變，繼續沿用原設計。



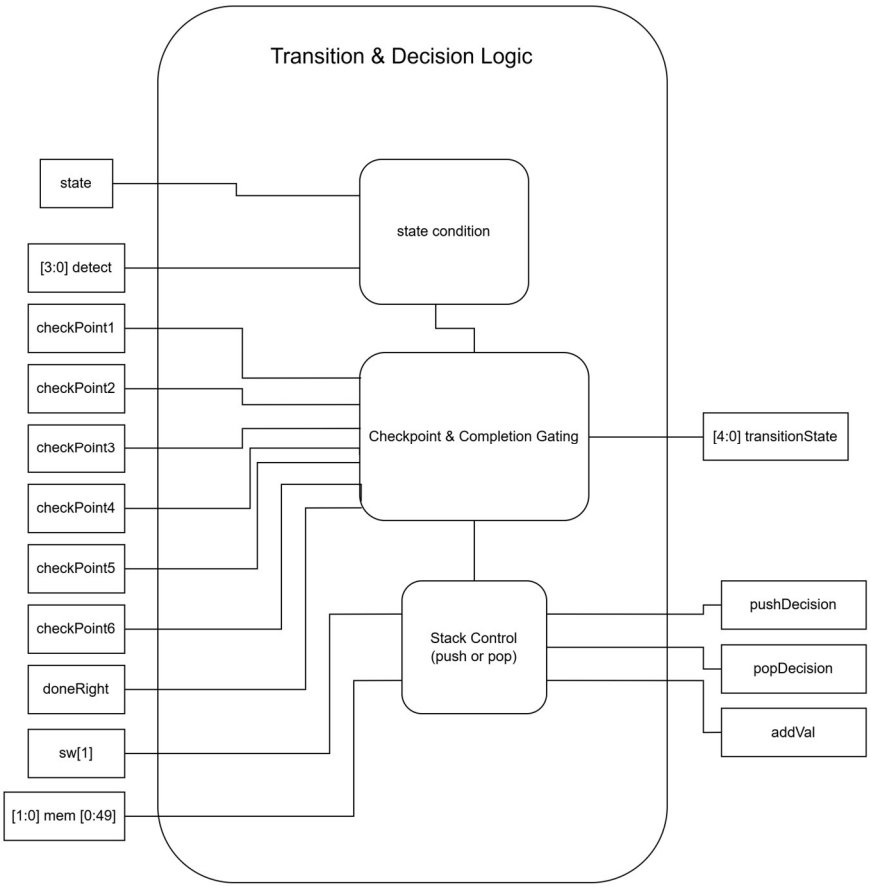
motor 根據傳入的 mode 決定馬達轉速及方向並轉換成對應波形。



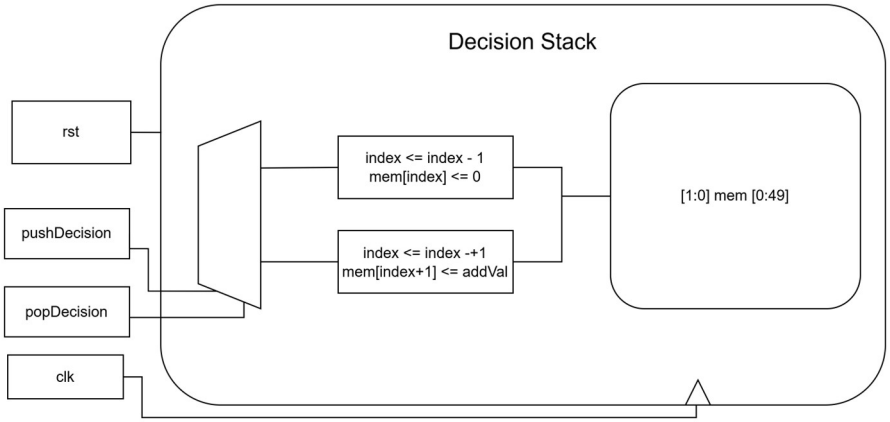
Checkpoint 目的是在解決 state transition 時，因車輛移動速度遠小於 state 更新速度，會造成在轉移到新的 state 時，探測器仍停在原先的地方尚未更新的問題，如停在路口(detect == 3b111)時，在 state update 後探測器仍在原先的路口，而造成後續判斷錯誤。



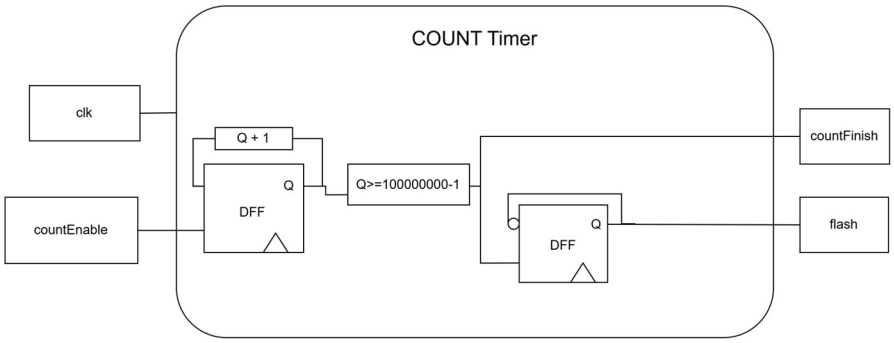
Transition Logic 是幫助 FSM 判斷 next state 的系統，因為在 FSM 進入 STOP state 後，會丟失 next state 的數據，這是因為 STOP state 是在兩個讓車輛往不同方向的 state 之間，為了讓馬達順利切換順逆轉而存在的，而 transition state 是為了記住在 STOP state 之後預計要進入的 state。

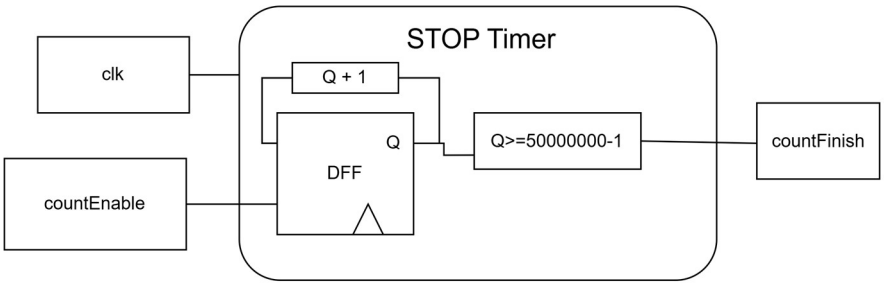


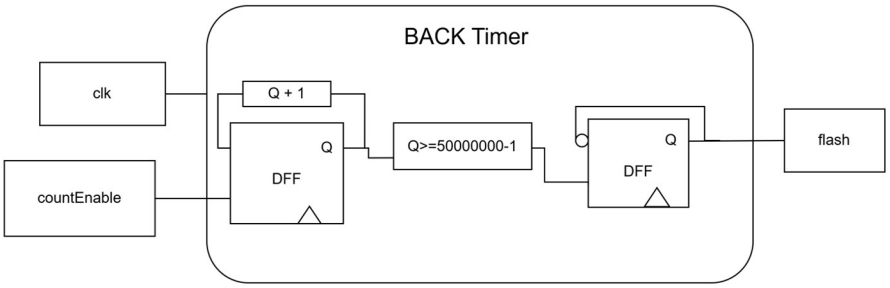
Decision Stack 是為了記錄車子在每個路口作出的方向選擇，以便在車子遇到死路需回退時能順利往不同方向繼續前進，包含 push 和 pop 兩種功能。

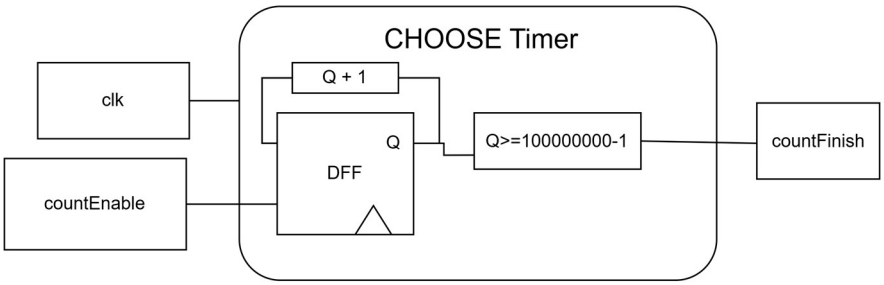


以下不同 Timer 能滿足前述提到的各式計時需求，根據不同功能，共有四種新增 Timer 加上課堂內提供之 clock_divider。









iv. Code Explanations

- Main Module Structure

```
module mainModule( ···
);
  // Parameter: FSM, Detect tracker
    // FSM ···
    // Detect tracker···
    // Mem···

  // Signal: wire, reg
    // System signal···
    // Sys.Counter signal···
    // IO signal···

  // Assign Block···

  // Circuit
    // System
      // Sys.Counter
        // Count Choose Time···
        // AUTO #turn right···
        // Led Display···
      // Sys.Signal/Mem Control
        // "CHECKPOINT":Sequential ···
        // "CHOOSE": Sequential···
        // "AUTO + Manual push/pop": Combinational···
        // "STACK": Sequential···
      // FSM
        // "STATE": Sequencial···
        // "NEXTSTATE": Combinational···
        // "TRANSITIONSTATE": Combinational···
    // IO
      // SevenSegment Display···
      // LED Display···
  // Module···
endmodule
```

- State Transition Design (NextState)

```
// "NEXTSTATE": Combinational
    always @(*)begin
        if(mode) nextState = FINISH;
        else begin
            case(state)
                IDLE: nextState = (sw[0])? START : IDLE;                    You, 1 minute
                START: nextState = (detect == 3'b010) ? COUNT : START;
                COUNT: nextState = (countFinish) ? STRAIGHT: COUNT;
                STRAIGHT:begin···
                end
                CHOOSE: begin ···
                end
                CHOOSE_DIR_STEP1:begin···
                end
                CHOOSE_DIR_STEP2:begin //choose···
                end
                CHOOSE_DIR_STEP3:begin···
                end
                LEFT:begin···
                end
                LITTLE_LEFT:begin//slightly fixing direction when going straight···
                end
                RIGHT:begin···
                end
                LITTLE_RIGHT:begin//slightly fixing direction when going straight···
                end
                BACK:begin···
                end
                STOP: nextState = (reSTART)? storeState : STOP;
                ERROR: nextState = (~sw[0]) ? IDLE : ERROR;
                default : nextState = state;
```

State 主要是引擎控制行為與道路判斷需求來做設計，其中包含負責主要轉向控制的 LEFT 與 RIGHT state，以及於直行過程中進行微小方向修正的 LITTLE_LEFT 與 LITTLE_RIGHT state；此外，為了判斷路口狀態，系統設計了 CHOOSE 與 CHOOSE_DIR_STEP1 ~ STEP3 ；在發生錯誤或需回溯時，則透過 BACK state 進行倒退修正，並利用 STOP state 調整引擎轉向與穩定車輛行為，使整體有限狀態機能在不同道路情境下維持穩定且有系統的控制流程。

```
STRAIGHT:begin
    case(detect)
    // ERROR STATE(0)

    // Transform state(2)
        ERROR_ROAD: begin
            // transitionState = BACK;
            nextState =  STOP;
        end
        TURN_ROAD111: nextState = (checkPoint2) ? CHOOSE : STRAIGHT;
    // Nothing Change(6)
        RIGHT_ROAD, RIGHT_LITTLE_ROAD: nextState = LITTLE_RIGHT;
        LEFT_ROAD, LEFT_LITTLE_ROAD: nextState = LITTLE_LEFT;
        TURN_ROAD101: nextState = STRAIGHT;
        STRAIGHT_ROAD: nextState = STRAIGHT;
        default : nextState = STRAIGHT;
    endcase
end
```

每個 state 中均會為所有可能偵測到的路況設定轉換邏輯，搭配 checkpoint 的設計避免多個 state transition 時因為遇到相同的轉換邏輯而出錯。

```
LEFT:begin···
end
LITTLE_LEFT:begin//slightly fixing direction when going straight
    case(detect)
    // ERROR STATE(0)

    // Transform state(2)
        ERROR_ROAD: begin
            // transitionState = BACK;
            nextState =  STOP;
        end
        TURN_ROAD111: nextState = (checkPoint2) ? CHOOSE : STRAIGHT;
    // Nothing Change(6)
        RIGHT_ROAD, RIGHT_LITTLE_ROAD: nextState = LITTLE_RIGHT;
        LEFT_ROAD, LEFT_LITTLE_ROAD: nextState = LITTLE_LEFT;
        TURN_ROAD101: nextState = STRAIGHT;
        STRAIGHT_ROAD: nextState = STRAIGHT;
        default :  nextState = LITTLE_LEFT;
    endcase
end
```

由於地圖/地面可能不太平整，以及引擎轉速等等不可控的因素，設計了 little left、little right 兩個 state，在偵測到直走時些微的左偏或右偏會自動切到這兩個 state 進行修正，避免因為地圖製作或是地面問題造成車子出線的狀況。

- State Transition Design (Transition State)

```
BACK:begin
    case(detect)
    // ERROR STATE(0)

    // Transform state(1)
        TURN_ROAD111: begin
            // transitionState = LEFT;
            nextState = STOP;
        end
    // Nothing Change(7)
        RIGHT_ROAD, RIGHT_LITTLE_ROAD:nextState = BACK;
        LEFT_ROAD, LEFT_LITTLE_ROAD:nextState = BACK;
        ERROR_ROAD: nextState = BACK;
        TURN_ROAD101: nextState = BACK;
        STRAIGHT_ROAD: nextState = BACK;
        default :  nextState = BACK;
    endcase
end
STOP: nextState = (reSTART)? storeState : STOP;
```

在特定情況下，車輛需進入 STOP state，使引擎暫停以完成轉向方向的切換；因此，系統必須記錄停止前的狀態，以確保車輛在結束 STOP state 後能正確回到原本預期的控制流程。為此，我們額外設計了一個 transitionState 變數，用以儲存即將轉換之目標狀態，並在 STOP state 結束後依據該變數進行狀態回復，使狀態轉換流程更加穩定且可預期。而 stop state 也會在短暫停止引擎運作後切入 transition state 變數紀錄的轉換 state 來維持系統的運作。

```
// "TRANSITIONSTATE": Combinational
    always @(*)begin
        transitionState = storeState;
        case(state)
            STRAIGHT, LITTLE_LEFT,LITTLE_RIGHT: begin
                transitionState = (detect == ERROR_ROAD) ? BACK : STRAIGHT;
            end
            CHOOSE: begin
                if(sw[1]) transitionState = (detect == TURN_ROAD101) ? CHOOSE_DIR_STEP1 : CHOOSE;
                else transitionState = (detect == TURN_ROAD101) ? LEFT : CHOOSE;
            end
            LEFT: begin···
            end
            RIGHT: begin···
            end
            BACK: begin  ···
            end
        endcase
    end
```

➢ Checkpoints Design

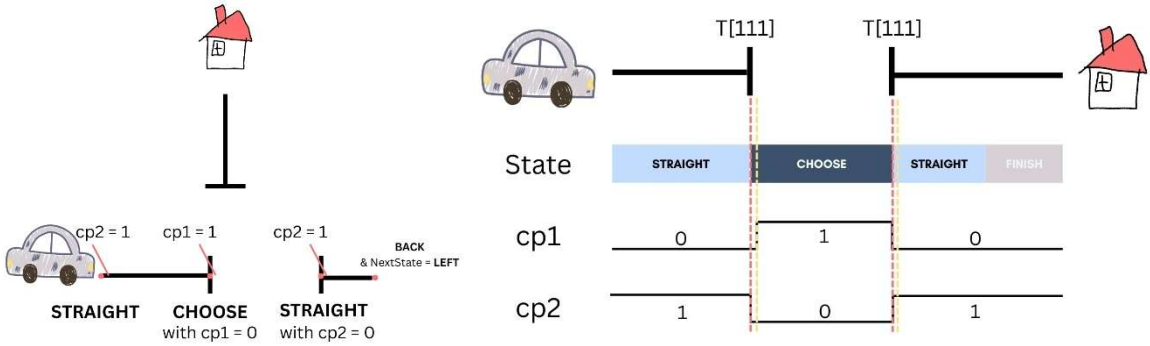
為了避免車子在偵測路線變換狀態時因為狀態轉換設計而出現錯誤轉換，我們設計了一套 checkpoint 機制來輔助有限狀態機（FSM）的決策流程。

```
// Checkpoint setting
    // cp1: from straight(111) to choose(111)
    // cp2: from left(111) to straight(111) && [straight(111) to choose(111)] first time
    // cp3: from stop(111) to left(111)
    // cp4: from stop(111) to right(111)
    // cp5: from choose(101) to left(101)
    // cp6: from choose(101) to right(101)
    reg checkPoint1,checkPoint2,checkPoint3,checkPoint4, checkPoint5,checkPoint6;
```

上圖註解為所有 checkpoint 需要負責偵測的跑道狀態轉換。

```
// "CHECKPOINT":Sequential
always @(posedge clk) begin
    if(rst) begin···
    end else begin
        storeState <= transitionState;
        // if(state == IDLE) begining <= 0;      You, yesterday • update code structure, with clear definition
        // straight -> choose(only detect 000)
        if(state == CHOOSE && detect != 3'b111)
            checkPoint1 <= 1;
        else if(state != CHOOSE)checkPoint1 <= 0;
        // left -> straight(may detect 010 or 000)
        if ((lastState == COUNT) || (((state == STRAIGHT) || (state == LITTLE_LEFT) || (state ==  LITTLE_RIGHT) ) && (detect !=
        3'b111))) checkPoint2 <= 1;
        else if(state != STRAIGHT && state != LITTLE_LEFT && state != LITTLE_RIGHT)checkPoint2 <= 0;
```

以 checkpoint 1, 2 舉例，如下圖所示：



Checkpoint 的設計原理是在特定 state 下，當感測器偵測到對應的跑道狀態時，將對應的 checkpoint 訊號拉高，以輔助 transition state 正確判斷狀態轉換條件。以 cp1 為例，其主要用於輔助由 CHOOSE state 轉換至 STRAIGHT state 的判斷流程；如上圖所示，當系統於 CHOOSE state 中首次偵測到非全黑跑道時，cp1 會被設為 1，在後續再次偵測到全黑跑道時，FSM 即可依據 cp1 的狀態順利完成由 CHOOSE state 至 STRAIGHT state 的轉換，其對應之狀態轉換邏輯程式碼所示(下圖)。

```
CHOOSE: begin
    case(detect)
    // ERROR STATE(0)

    // Transform state(2)
        TURN_ROAD101: begin
            // transitionState = LEFT;
            nextState = STOP;
        end
        TURN_ROAD111:begin//nextState = (checkPoint1)? STRAIGHT : CHOOSE;
            if(sw[1])begin
                nextState = (checkPoint1)? CHOOSE_DIR_STEP1 : CHOOSE;
            end
            else begin
                nextState = (checkPoint1)? STRAIGHT : CHOOSE;
            end
        end
```

- Stack Design

在課程提及的 memory 處理方面，資料存入及讀取往往會相差一個 clock edge，這對於車輛需要及時 push 和 pop 資料進 stack 以及讀取 stack 會需要較麻煩的處理，因此我們並沒有把 stack 獨立成一個 module，而是直接放進主程式，並用 one-pulse 過的 push 及 pop 訊號模擬 stack 的運作，而該兩種訊號是在 combinational block 裡判定的，這是為了避免一個 clock edge 造成的誤差而不使用 sequential block。下圖展示了 stack 的初始化、push 及 pop 的 one-pulse 處理以及兩種操作的具體細節。

```
always@(posedge clk,posedge rst)begin
    if(rst)begin
        index <= 0;
        for(rst_index = 0; rst_index <= 49; rst_index = rst_index + 1)begin
            mem[rst_index] <= 2'b00;
        end

        pushDecision_s <= 0;
        popDecision_s <= 0;
    end
    else begin
        pushDecision_s <= pushDecision;
        popDecision_s <= popDecision;
        if(!pushDecision_s && pushDecision && !popDecision_s && popDecision)begin
            mem[index] <= addVal;
        end
        else if(!pushDecision_s && pushDecision)begin
            if(index < MEM_DEPTH - 1)begin
                mem[index + 1] <= addVal;
                index <= index + 1;
            end
        end
        else if(!popDecision_s && popDecision)begin
            if(index > 0)begin
                mem[index] <= 2'b00;
                index <= index - 1;
            end
        end
    end
end
```

- AUTO Mode Logic

自動模式中，車子首先會直走，接著左轉，最後是右轉，若直走失敗需選擇左轉或右，需依照 stack top 做決定。

```
if(detect == 3'b111 && mem[index] == DEC_STRAIGHT)begin
        transitionState = LEFT;
end else if(detect == 3'b111 && mem[index] == DEC_LEFT)begin
        transitionState = RIGHT;
end else transitionState = BACK;
```

- Manual Design (Extra Choose)

  為了盡可能銜接 AUTO mode 之結構，我們並不重新設計 CHOOSE state，而是在 CHOOSE state 之後加入新的三個 state：CHOOSE_DIR_STEP1、2、3。原先直走及左右轉的 state transition 並不相同，在一個路口時，車子左右轉相比直走需要多出後退的步驟，因輪子順逆轉的變換需求，中間要經過更多 STOP state 以及 transition state 的判斷，故新增三個 state 可以直接完成車輛停止與輪子反轉的需求，銜接後退及左右轉，既不讓原有 code 更為複雜，也讓手動與自動之間的切換更具彈性，且未來若要在手動模式擴增其他功能，也無須動到 AUTO mode 的架構，僅需於此三個新 state 之間處理即可，如下圖，在 CHOOSE_DIR_STEP1 先判斷車子要直走還是左右轉，若選直走則 next state 為 STRAIGHT，如此一來與 AUTO mode 直走邏輯即相同。若需左右轉，則需先讓車子倒退再轉向，倒退由 CHOOSE_DIR_STEP2 完成，順逆轉變換由 CHOOSE_DIR_STEP3 產生之一個 clock edge 的間隙完成，並銜接左或右轉。

```
CHOOSE_DIR_STEP1:begin
        if(countCheck)begin
            case(joyStickDir)
            2'b00: begin
                nextState = STRAIGHT;
            end
            // 2'b01:chosenState = RIGHT;
            // 2'b11:chosenState = LEFT;
            default:nextState = (sw[1])? CHOOSE_DIR_STEP3 : CHOOSE_DIR_STEP2;
            endcase
        end
        else nextState = CHOOSE_DIR_STEP1;
end
CHOOSE_DIR_STEP2:begin

    if(detect != 3'b111 && detect != 3'b101)nextState = CHOOSE_DIR_STEP3;
    else nextState = CHOOSE_DIR_STEP2;
end
CHOOSE_DIR_STEP3:begin
    nextState =  chosenState;
end
```

。

III. 實作/ 難易度 / 分工

i. 實做完成度：

我們有對 proposal 進行修改。最初的構想是連接兩片開發板，並透過無線方式進行互動；然而在討論過程中發現，藍牙模組在使用上存在諸多限制，且線路連接與干擾問題可能影響感測器的穩定性與判斷準確度。

因此，我們調整為第二個版本的設計，改以杜邦線直接連接車子與搖桿。由於杜邦線重量較輕，且可彈性延長線路長度，使得整體控制更為穩定，也能有效降低感測誤差，在實作與操作上皆具有較佳的表現。

最終專案整體完成度約達原先預期之 90%。目前仍有部分細部設計與判斷邏輯尚未完全完善，受限於既有系統架構之設計限制，暫時尚未提出可行的解決方案。

ii. 難易度：

- 使用的額外模組幾乎都從本學期課程中的 sample code 中取得，額外 pmod 亦有相關 sample code 可以參考並且修改。唯 main module 的設計以及判斷

iii. 分工：

- 在 Git repository 的 /Verilog/UpdateLog.md 中，詳細記錄了各階段的工作時程與實作進度。
- 本專題之程式碼幾乎皆由組員共同協作完成並進行測試，下表僅標註部分系統架構中負責主要設計與撰寫的成員。

	江佩霖	劉廷暉
FSM		
IO / Map		
Pmod		
Stack		
Manual mode		
Implementation		

IV. 是否包含課程外部分以及比重

- 課外延伸內容僅包含搖桿（joystick）實作的部分，大約佔整體專題內容的 5%，其餘皆屬於課程中所涵蓋之範圍。
- 搖桿的 Sample Code 取自 Digilent 官網，在 /Verilog/SampleCode 中有詳細的標註以及下載的原始代碼。

V. 測試完成度

- 所有測試均有錄影在 /Verilog/TestVideo 資料夾中。

	AUTO	Manual
Basic	✓	× (不支援此模式)
Adv1	✓	✓
Adv2	✓	✓

VI. 困難與解決方法

i. 車子尺寸與地圖設計

由於車體本身尺寸較大，我們在轉彎機制上採用「一輪前轉、一輪後轉」的方式，縮小車

子轉彎所需的空間。能有效降低轉彎區域的占用面積，提升地圖設計的彈性。

ii. 引擎穩定性

在實際測試過程中發現，右輪引擎的轉速不穩定，且時常高於左輪，導致車輛在直線行進時容易偏移，轉彎時亦可能出現突發加速的情況，增加了地圖設計與行為預測的困難度。由於除了更換引擎外，較難透過硬體方式解決此問題，因此我們微調了程式中左右輪的轉速設定，使其盡量趨於一致，並在地圖設計上配合車輛實際行徑特性，適度調整走道配置。

iii. 不同操作模式的切換設計

在撰寫程式碼時，我們選擇先完成自動模式（AUTO）的核心邏輯，再加入手動模式（Manual）的設計。由於系統可於兩種模式間切換，原有程式碼中的部分判斷條件無法直接沿用，且若直接重新為新的模式撰寫邏輯會導致程式碼攏長不易讀。因此我們重新規劃部分訊號連線與模組設計，使兩種模式的程式邏輯彼此不衝突，並最大化共用模組的使用，維持程式架構的清楚與精簡。

iv. 搖桿控制設計

專案有額外加入搖桿提升手動操作的便利性，但由於搖桿需透過實體線路與開發板連接，在操作過程中仍可能對車輛行進造成影響；此外，若於搖桿訊號傳輸後立即進行 state 切換，會使上述影響更加明顯。且原始搖桿範例設計主要以橫向操作為主，若直接使用在我們的專案中，操作上較不直覺。

因此有重新設計了搖桿的操作方向，改為直拿方式使用，使使用者能更精準地控制方向並清楚確認按鈕狀態。同時在訊號處理上加入適當的延遲機制，使用者在完成操作後，有足夠時間將搖桿放回車體上，降低搖桿連接對車輛行進所造成的干擾。

v. 不同 state 間重複判定問題

由於系統所使用的 detector 最多僅能判斷 8 種不同的跑道組合，在初期設計 state 與地圖時，便會不可避免地出現重複判定的情況。舉例來說，從狀態 A 轉換至 B 是透過偵測到三個黑色跑到進行判斷，而從 B 轉換至 C 亦可能使用相同的判定條件，導致狀態轉換不易區分。此問題在使用者輸入模式下更為明顯，因轉換情境更加多樣。針對上述情況，我們於系統中設計多個 checkpoint 作為輔助判斷條件，以降低重複判定造成的影響。儘管受限於 detector 可判斷的組合數量，仍無法完全解決使用者輸入模式中較特殊的重複轉換問題，但我們於後續調整地圖設計，盡可能避免產生相同的偵測條件，以最大化降低重複判定發生的機率。

vi. Stack 設計

為了能正確記錄 AUTO 模式中車子行走時所經過的路口狀況，我們自行設計了一個 stack 資料結構。在每個需要轉彎的路口，會將車子的判斷結果推入 stack；若之後發現判斷錯誤，則根據 pop 出來的結果，決定下一個嘗試的轉彎方向，並調整引擎的轉動邏輯。在設計此資料結構時，除了需要特別注意 push 與 pop 的時機與順序外，也必須妥善維護目前 stack 的頂端 index，否則容易造成資料錯位或路徑回溯錯誤，進而影響系統整體的判斷與行走正確性。

VII. 心得討論

111062118 江佩霖

- 在實作過程中，我認為最難的地方是引擎的狀態，雖然相關控制參數是固定的數值，但在實際測試時，車輛行為仍會因環境與狀態切換的細微差異而產生不一致，甚至偶爾出現暴衝現象，提高了地圖設計與測試的難度。還好較早開始進行專題規劃與實作，整體時程安排也比較平均，能在最後階段完整地完成系統整合與測試。雖然在 demo 時出現些許突發狀況，但整體專題過程中，無論是在 Git 版本控制的使用，或是團隊成員間的協作與溝通上，都獲得很多的實務經驗；尤其在電路設計與程式碼整合過程中，由於成員的 coding style 不太相同，經過反覆的修改與討論，不僅提升了程式與電路邏輯的穩定度，也促進了彼此在系統設計能力上的成長。
- 唯一稍微比較遺憾的地方是我們在硬體擴充方面比較保守，僅使用搖桿作為使用者輸入介面，沒有額外加入更多感測器或影像設備；不過對於地圖的設計以及 FSM state 的討論與設計也一起花了很多心思，相信這個專案的完成花費的時間以及心力並沒有輸給其他組別！

113062221 劉廷暉

- 車子相對其他主題的困難點在於需花費大量心力做微調以及地圖設計，為了找出失敗點是在於車子還是地圖，需額外花心思 debug。在地圖上，為了配合只有三個探測器的車子，我們在地圖設計上花費不少心思，迷宮路口設計已經大致是探測器的極限。我認為我們在視覺效果上雖不如其他組，但對於車子的發展是有所突破的，若未來有更加穩定的馬達及更先進的探測器，我相信我們能做的更多、更深且更好，唯一可惜的是我們因車輛及時間限制未能把程式的擴充性發展到最好，否則應可實現更多功能。
- 我們在團隊分工上也做得很好，由於只有一台車子，在 debug 及測試上都須良好

  的時間規劃及安排，這讓我學到很多!