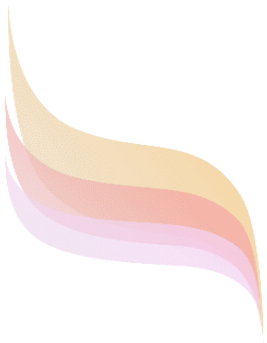
# Digital System Design

## Lecture 10
## Design Examples II

## Learning Examples:

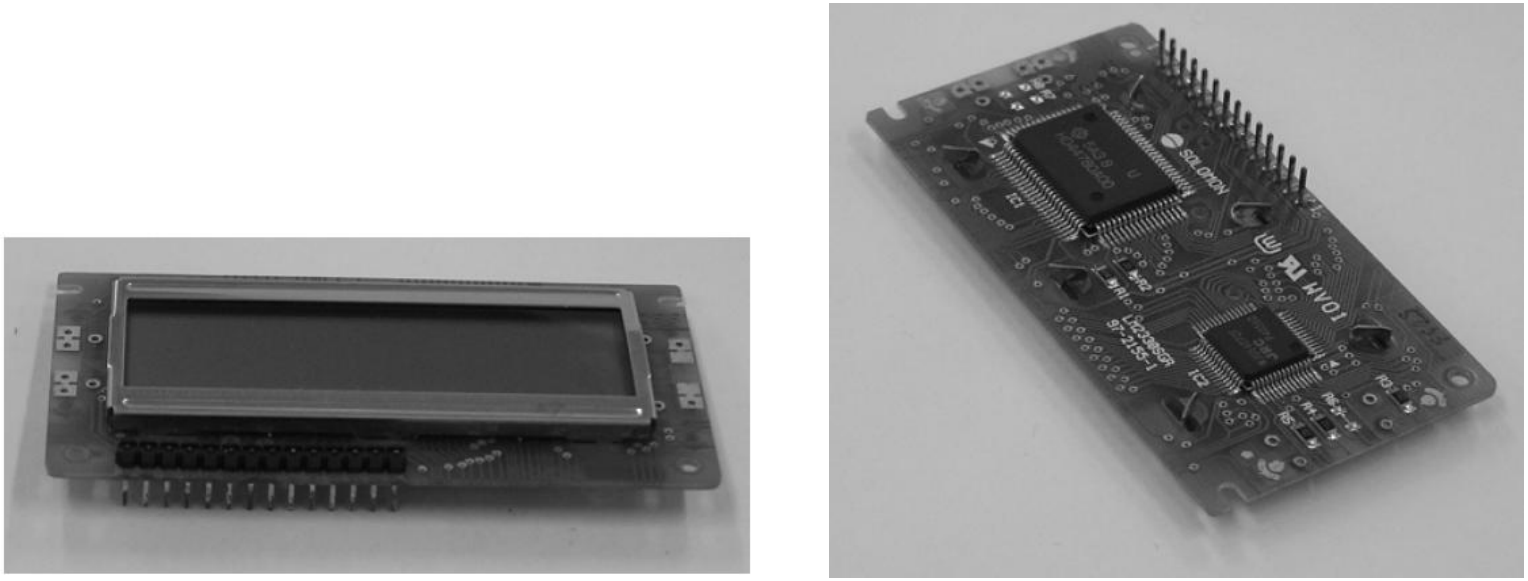
- LCD螢幕顯示
- PS/2 Keyboard
- VGA螢幕顯示

# LCD 螢幕顯示



# 認識LCD模組 (LCM)

**LCD**(Liquid Crystal Display)爲液晶顯示面板，由於 LCD 的控制須專用的驅動電路，且 LCD 面板的接線須特殊技巧，加上 LCD 面板結構較脆弱，通常不會單獨使用。而是將 LCD 面板、驅動與控制電路組合而成一個 LCD 模組(Liquid Crystal Display Moulde，簡稱爲 **LCM**)。LCM 是一種很省電的顯示裝置，常被應用在數位或微電腦控制的系統，做爲簡易的人機介面



LCD 模組(左邊為正面圖、右邊為背面圖)

# LCM基本資料

LCM 的種類煩多，而在學校與訓練單位所採用的 LCM，大都是以日商日立公司的控制器(HD44780)所組成的 LCM，其特性如下：

- 內建 80bytes 資料顯示記憶體(Data Display RAM，簡稱 **DD RAM**)，可顯示 16 字×1 列、20 字×1 列、16 字×2 列、20 字×2 列、40 字×2 列等模式。
- 內建字型產生器(Character Generate ROM，簡稱 **CG ROM**)，可產生 160 個 5×7 字型。
- 自建字型產生器(Character Generate RAM，簡稱 **CG RAM**)，可由使用者自建 8 個 5×7 字型。

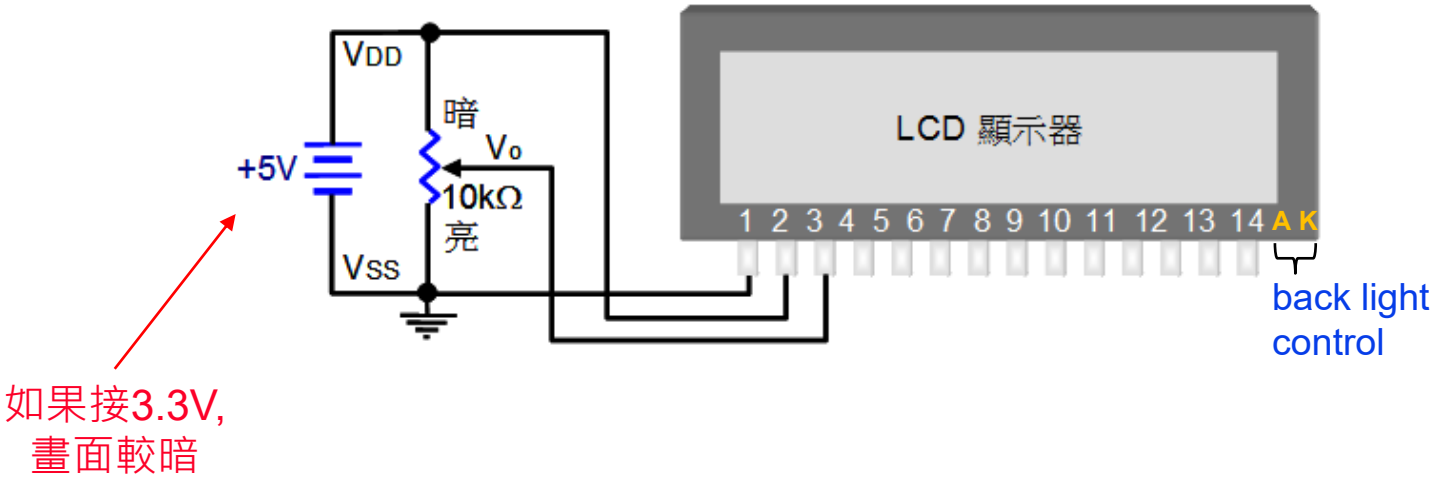

**2列 x 16字**

| | C1 | C2 | C3 | C4 ~ | C16 |
|---|---|---|---|---|---|
| R1 | 80H | 81H | 82H | 83H ~ | 8FH |
| R2 | C0H | C1H | C2H | C3H ~ | CFH |

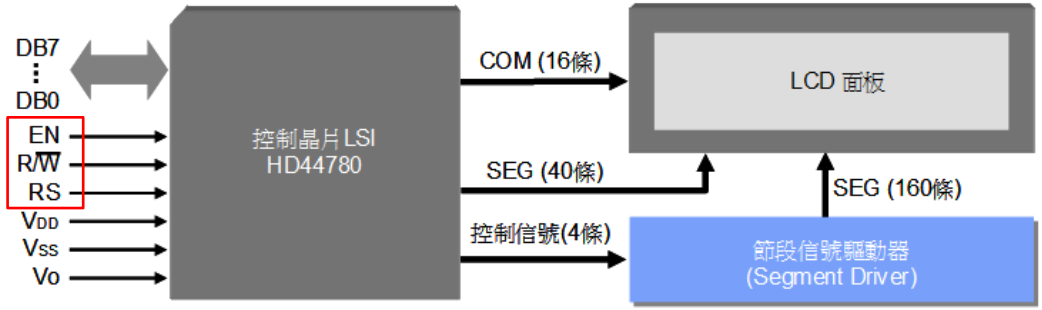
# 電源接腳

- 電源接腳：$V_{DD}$、$V_{SS}$、$V_O$

LCM 有三支電源接腳，一為+5V 電源 $V_{DD}$，一為接地線 $V_{SS}$，另一為 LCM 驅動電源 $V_O$。如圖 12-2 所示 LCM 電源接線圖，$V_O$ 可由 $V_{DD}$ 與 $V_{SS}$ 間的電壓分壓取得，當電壓 $V_O$ 愈小時，LCM 明暗對比愈強，當 $V_O$ 電壓愈大時，LCM 明暗對比愈弱。



# LCM腳位

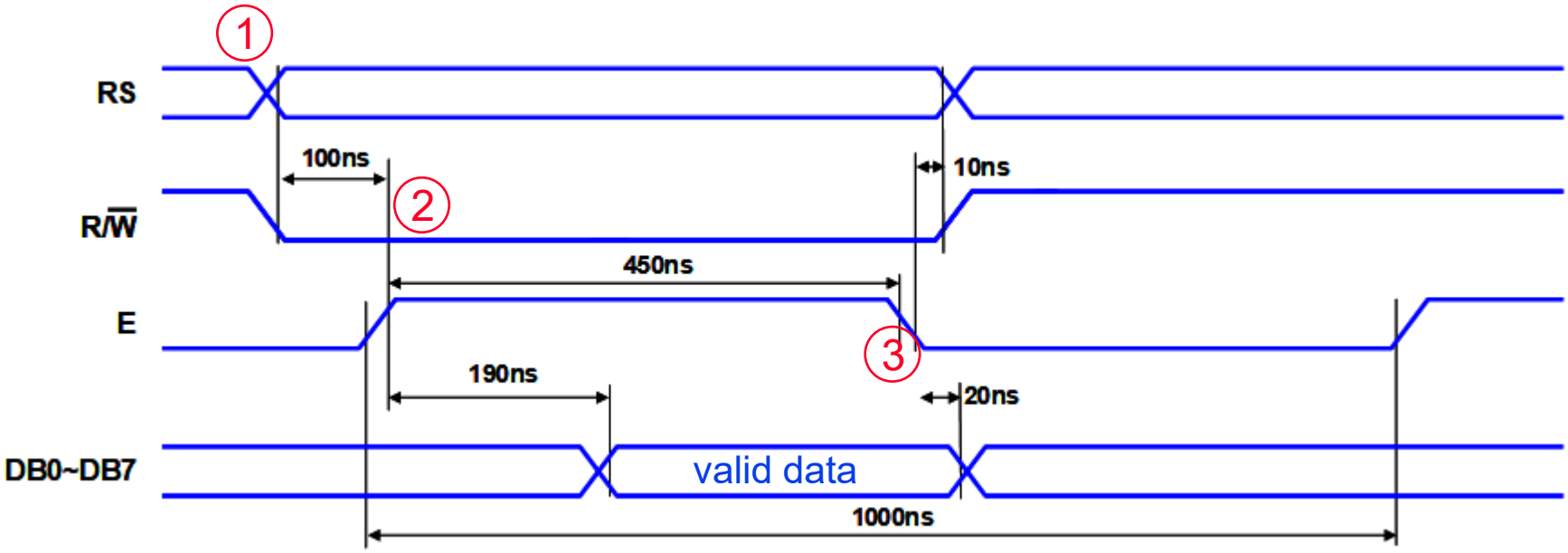



| | | |
|---|---|---|
| $V_{SS}$ | I | 接地腳。 |
| $V_{DD}$ | I | +5V 電源。 |
| $V_O$ | I | 顯示明暗對比控制。 |
| RS | I | RS=0，選擇指令暫存器，RS=1，選擇資料暫存器。 |
| R/$\overline{W}$ | I | R/$\overline{W}$ =0，將資料寫入 LCM 中；R/$\overline{W}$ =1，自 LCM 讀取資料。 |
| EN | I | 致能（enable）LCM 動作。 |
| DB0 | I/O | 資料匯流排 (LSB) |
| DB1 | I/O | 資料匯流排 |
| DB2 | I/O | 資料匯流排 |
| DB3 | I/O | 資料匯流排 |
| DB4 | I/O | 資料匯流排 |
| DB5 | I/O | 資料匯流排 |
| DB6 | I/O | 資料匯流排 |
| DB7 | I/O | 資料匯流排 (MSB) |

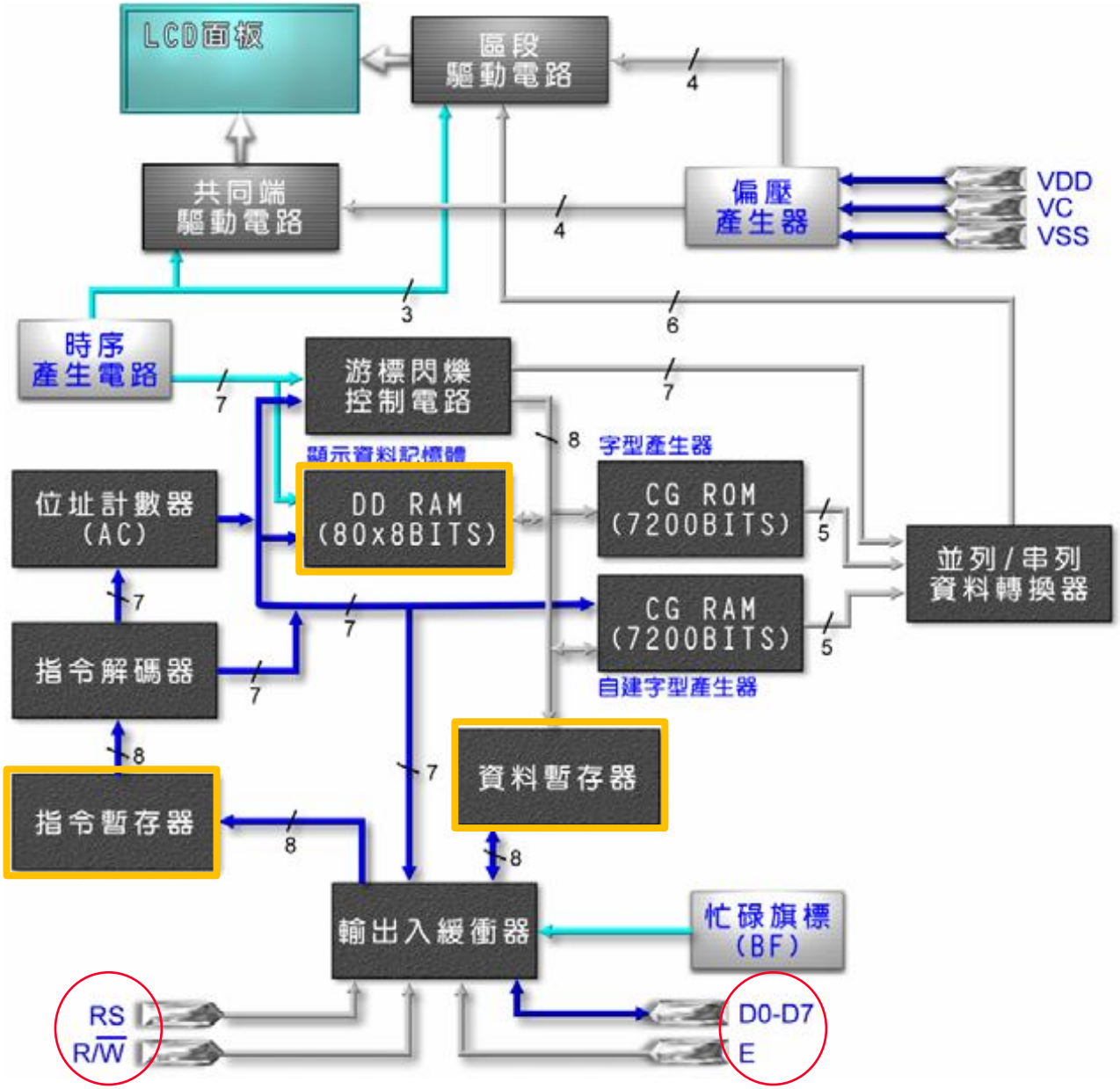
# LCM Timing

- 寫入時序

1. RS=0 時表示寫指令至 LCM 中；RS=1 時表示寫資料至 LCM 中。
2. 因為是寫入動作，R/$\overline{W}$=0，再致能 LCM（EN=1）
3. 然後 指令或資料寫入 LCM 匯流排中，最後必須將 EN=0。 -> to latch in data



# HD44780 LCM 內部結構



# CG ROM字型

這個字型編碼表其實就是ASCII 編碼。

# LCM的指令集

| 指令 | 控制訊號 | | 指令碼(Data Bus上的資料) | | | | | | | | 執行 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 功能 | RS | R/W | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | 時間 |
| 清除顯示器 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1.64ms |
| 游標歸位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | * | 1.64ms |
| 進入模式 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | I/D | S | 40us |
| 顯示器開 / 關 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | 40us |
| 顯示器移位 | 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | * | * | 40us |
| 功能設定 | 0 | 0 | 0 | 0 | 1 | DL | N | F | * | * | 40us |
| 設定CG ram位址 | 0 | 0 | 0 | 1 | CG RAM 的位址 | | | | | | 40us |
| 設定DD ram位址 | 0 | 0 | 1 | DD RAM 的位址 | | | | | | | 40us |
| 讀取忙碌旗標 | 0 | 1 | BF | 位址計數器 | | | | | | | 40us |
| 寫入資料 | 1 | 0 | 寫入的資料 | | | | | | | | 40us |
| 讀取資料 | 1 | 1 | 讀取的資料 | | | | | | | | 40us |

CG RAM：是一個可以由使用者自行設定字型的地方，LCM共有64個位元組的CG RAM使用者可以在這些方自行造字，一般而言一個顯示位置，對應到CG RAM的八個位元組，所以使用者可自行造八個字型。

DD RAM：LCM上不管那種文字型的LCM，其內部提供80個位元組的顯示記憶體，使用者可以將欲顯示的料放入這些位置上，就會顯示在相對應的顯示幕上。

# LCM控制指令 - 1

LCM 控制指令只有11 個，依序介紹這下：

## 1．清除顯示(clear display)

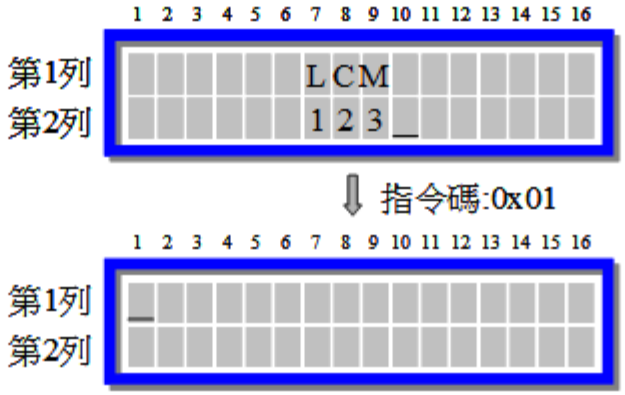
| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

(1) 目前 DDRAM 的內容將會全部填入 0x20（空白字元）。

(2) 游標移到第 1 列第 1 行位置。

(3) 清除位址計數器（address counter，簡記 AC）的內容，即 AC=0。

- 指令範例：



# LCM控制指令 - 2

## 2．游標歸位(return home)

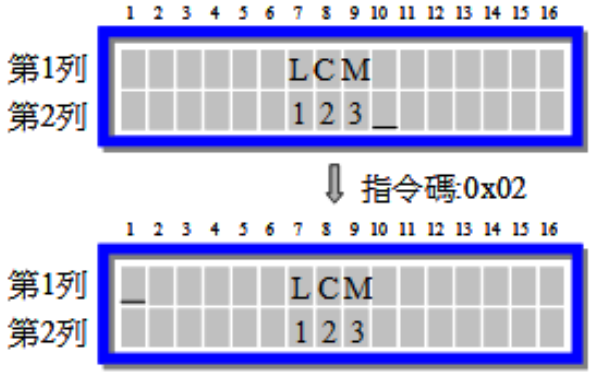
| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | × |

(1) 目前 DDRAM 的內容保持不變。

(2) 游標移到第 1 列第 1 行位置。

(3) 清除位址計數器的內容清除，即 AC=0。

- 指令範例：



# LCM控制指令 - 3

## 3．輸入模式設定(entry mode set)

| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | I/D | S |

輸入模式設定

| 位元 1 (I/D) | 位元 0 (S) | 動作說明 |
|---|---|---|
| 0 | 0 | 顯示字元不動，游標左移，AC 值減 1。 |
| 1 | 0 | 顯示字元不動，游標右移，AC 值加 1。 |
| 0 | 1 | 顯示字元右移，游標不動，AC 值不變。 |
| 1 | 1 | 顯示字元左移，游標不動，AC 值不變。 |

(1) I/D 位元是控制 AC 值遞增或遞減。當 I/D=1 時 AC 值遞增（increment），當 I/D=0 時 AC 值遞減（decrement）。

(2) S 位元在控制顯示字元的移位與否。當 S=1 則字元移位（shift），當 S=0 則字元不動。

# LCM控制指令 - 4

## 4．顯示器控制(display ON/OFF control)

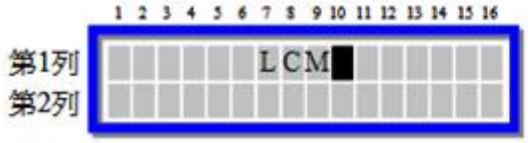
| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B |

顯示器控制

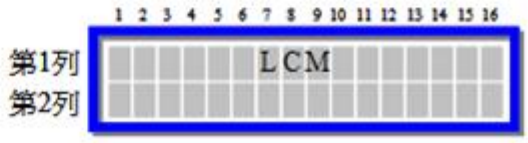
| 圖 12-6 元 | 圖 12-7 英文名稱 | 圖 12-8 中文名稱 | 圖 12-9 動作說明 |
|---|---|---|---|
| 2 | D | 顯示（Display） | D=0:關閉顯示器，D=1:開啟顯示器。 |
| 1 | C | 游標（Cursor） | C=0:不顯示游標，C=1:顯示游標。 |
| 0 | B | 閃爍（Blink） | B=0:游標不閃爍，B=1:游標閃爍。 |

(1) 正常情形下，設定 D=1，C=1，B=0 開啟顯示器，顯示游標但不閃爍。

(2) 設定 B=1 時的游標閃爍情形如圖 12-6 所示。




(a) 游標閃爍且顯示時




(b) 游標閃爍且不顯示時

# LCM控制指令 - 5

## 5．游標移位控制(cursor and display shift)

| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | × | × |

游標移位控制

| 位元 3 (S/C) | 位元 2 (R/L) | 動作說明 |
|---|---|---|
| 0 | 0 | 游標左移，AC 值減 1。 |
| 0 | 1 | 游標右移，AC 值加 1。 |
| 1 | 0 | 整個顯示幕向左移動。 |
| 1 | 1 | 整個顯示幕向右移動。 |

(1) S/C 位元是在控制游標或整個顯示幕的移動。當 S/C=1 時則顯示幕移位（display shift），當 S/C=0 時則游標移動（cursor move）。

(2) 本指令不會改變 DD RAM 資料。

# LCM控制指令 - 6

## 6．功能設定(function set)

| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | DL | N | F | × | × |

功能設定

| 位元 | 英文名稱 | 中文名稱 | 動作說明 |
|---|---|---|---|
| 4 | DL | 資料長度<br>（data length） | DL=0:4 位元（DB7~DB4）<br>DL=1:8 位元（DB7~DB0） |
| 3 | N | 列數（Number） | N=0：1 列顯示；N=1：2 列顯示。 |
| 2 | F | 字型（Font） | F=0：5×7 字型；F=1：5×10 字型。 |

(1) DL 位元是在控制傳送資料位元長度。DL=1 時使用 8 位元資料匯流排（DB7~DB0）。DL=0 時使用 4 位元的資料匯流排（DB7~DB4），先送高四位元資料，再送低四位元資料，適用於較少 I/O 腳的微控制器。DL 預設值為 1。

(2) N 位元在設定顯示字型的列數，當 N=0 時為 1 列顯示，當 N=1 時為 2 列顯示，只能顯示 5×7 字型，因此 F 的設定值無效。N 預設值為 0。

(3) F 位元在設定顯示字型，當 F=0 時顯示 5×7 字型，當 F=1 時顯示 5×10 字型。F 預設值為 0。

# LCM控制指令 - 7

## 7．CGRAM 位址設定(set CG RAM address)

| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | A5 | A4 | A3 | A2 | A1 | A0 |

(1) 本指令在設定 CGRAM 位址，接著輸入的資料將會寫入 CGRAM 中。

(2) LCM 提供使用者可以自建 8 個 5×7 字型，字型碼為 0x00~0x07。

# LCM控制指令 – 8, 9

## 8．DDRAM 位址設定(set DD RAM address set)

| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |

(1) 本指令在設定 DDRAM 位址，接著輸入的資料將會寫入到 DDRAM 中。

(2) LCM 第 1 列第 1 行 DDRAM 位址為 0x00，再加上 DB7=1 則實際位址為 0x80。
LCM 第 2 列第 1 行 DDRAM 位址為 0x40，再加上 DB7=1 則實際位址為 0xC0。

## 9．讀取忙碌旗標 BF 及位址計數器 AC 內容

| RS | R/$\overline{W}$ | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | BF | A6 | A5 | A4 | A3 | A2 | A1 | A0 |

(1) 當忙碌旗標 BF=1 時，表示 LCM 正在處理資料，不可再寫入資料，當 BF=0 時，表示 LCM 處於閒置（idle）狀態，可以再寫入資料。

(2) AC 值為最近設定的 RAM 實際位址（CG RAM 或 DD RAM）。

# LCM控制指令 – 10, 11

## 10．將資料寫入 DDRAM 或 CGRAM 中

| RS | R/$\overline{W}$ |
|---|---|
| 1 | 0 |

| DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

(1) 將資料寫入 DDRAM 或 CGRAM 之前，須先設定 DDRAM 或 CGRAM 的位址。

(2) 如果設定 DDRAM 位址，則本指令會將資料寫入 DDRAM 中，如果設定 CGRAM 位址，則本指令會將資料寫入 CGRAM 中。

## 11．自 DDRAM 或 CGRAM 讀取資料

| RS | R/$\overline{W}$ |
|---|---|
| 1 | 1 |

| DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

(1) 自 DDRAM 讀取資料前，須先設定 DDRAM 的位址。

(2) 自 CGRAM 讀取資料前，須先設定 CGRAM 的位址。

# 8位元模式的初始設定範例

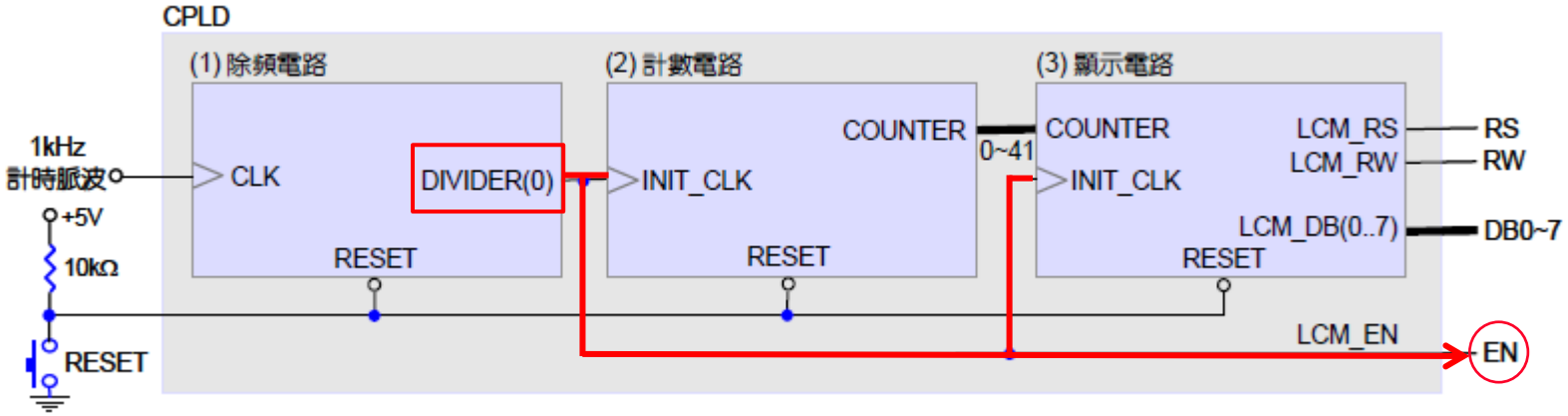
# Example 1: Print Strings on LCD ( lcd_cgrom.vhd )

## ➢ Using a counter

**功能說明：**

使用 CPLD 晶片，設計 LCM 顯示內建字型的數位電路。若按下 RESET 鍵，LCM 顯示如圖 12-11 所示。


圖12-11　LCM 顯示內建字型電路的顯示情形




Source: CPLD 數位邏輯設計實務

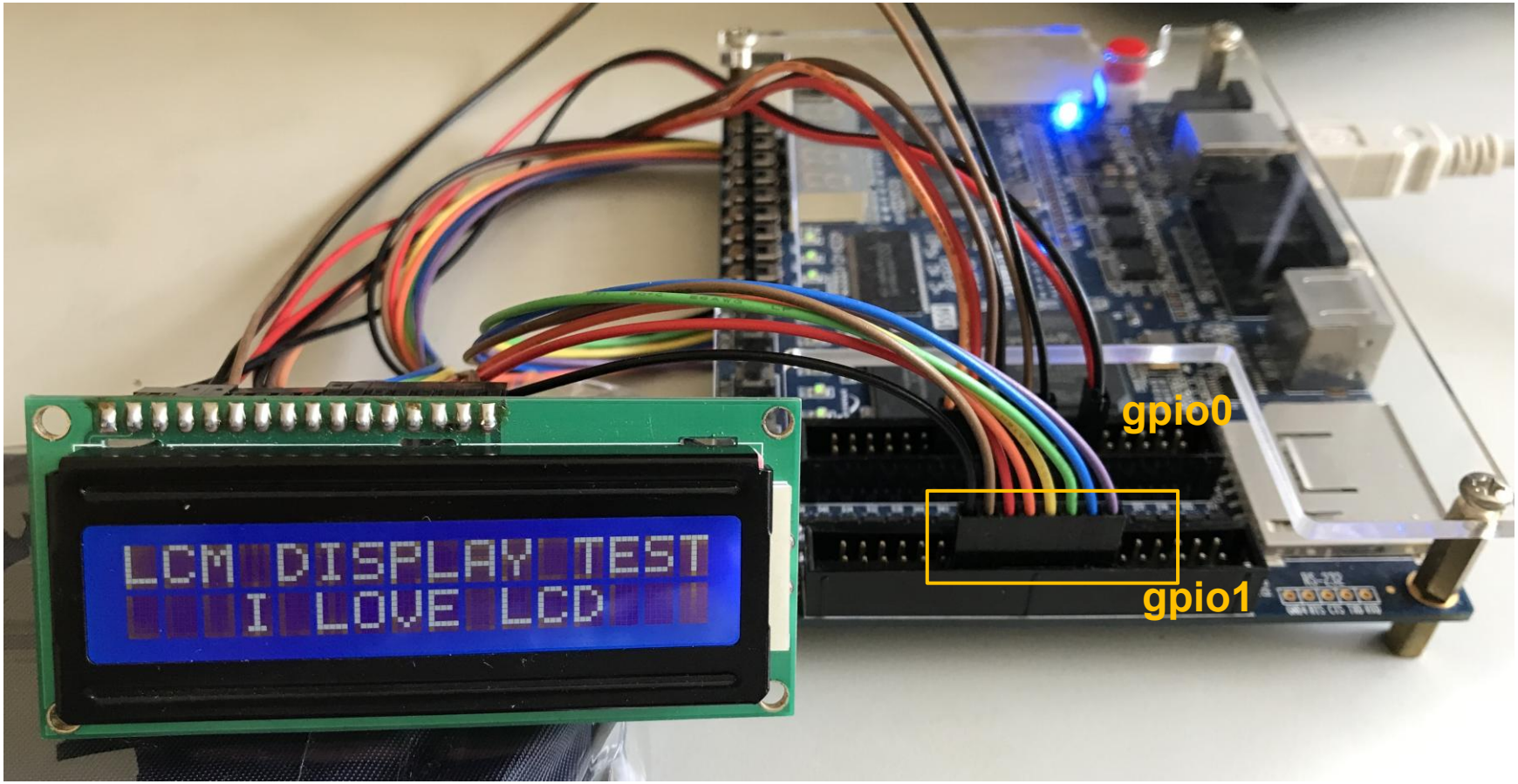
# Example 1: Print Strings on LCD ( lcd_cgrom.vhd )

```
ENTITY LCD_Display IS
PORT ( clk, reset: IN STD_LOGIC;
       RS, RW, EN: OUT STD_LOGIC;
       DB : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) );
END LCD_Display;
```

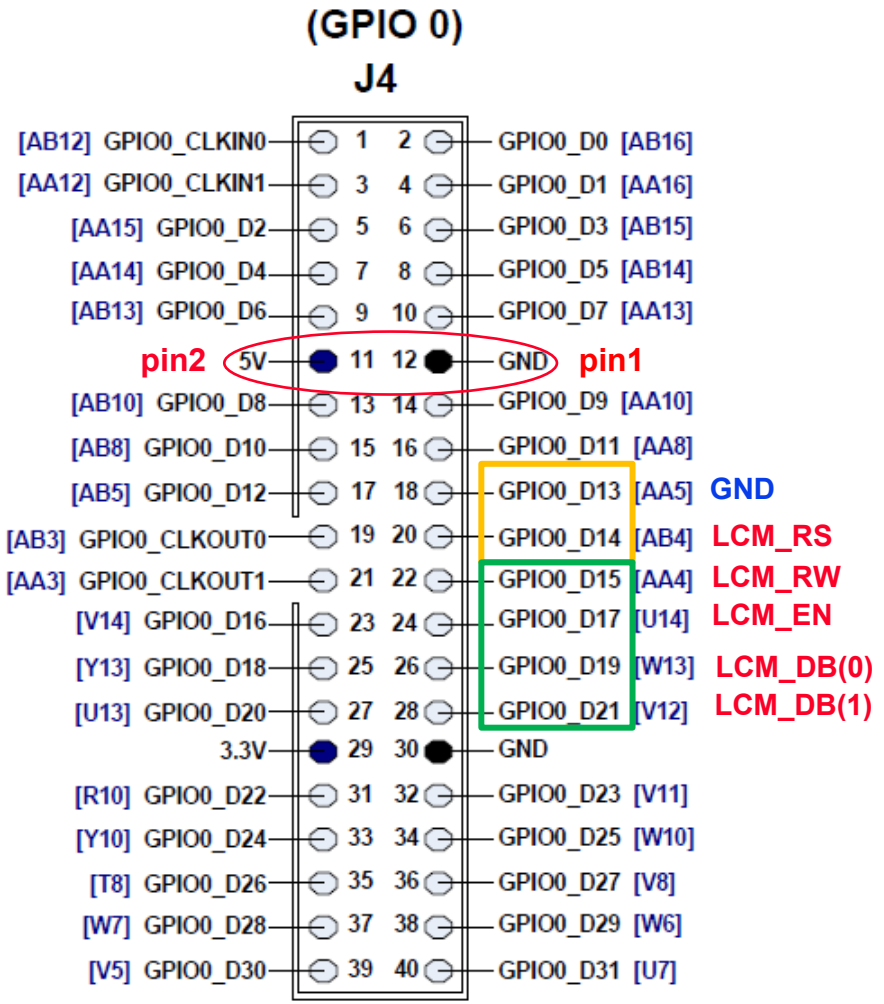
Implementation on DE0

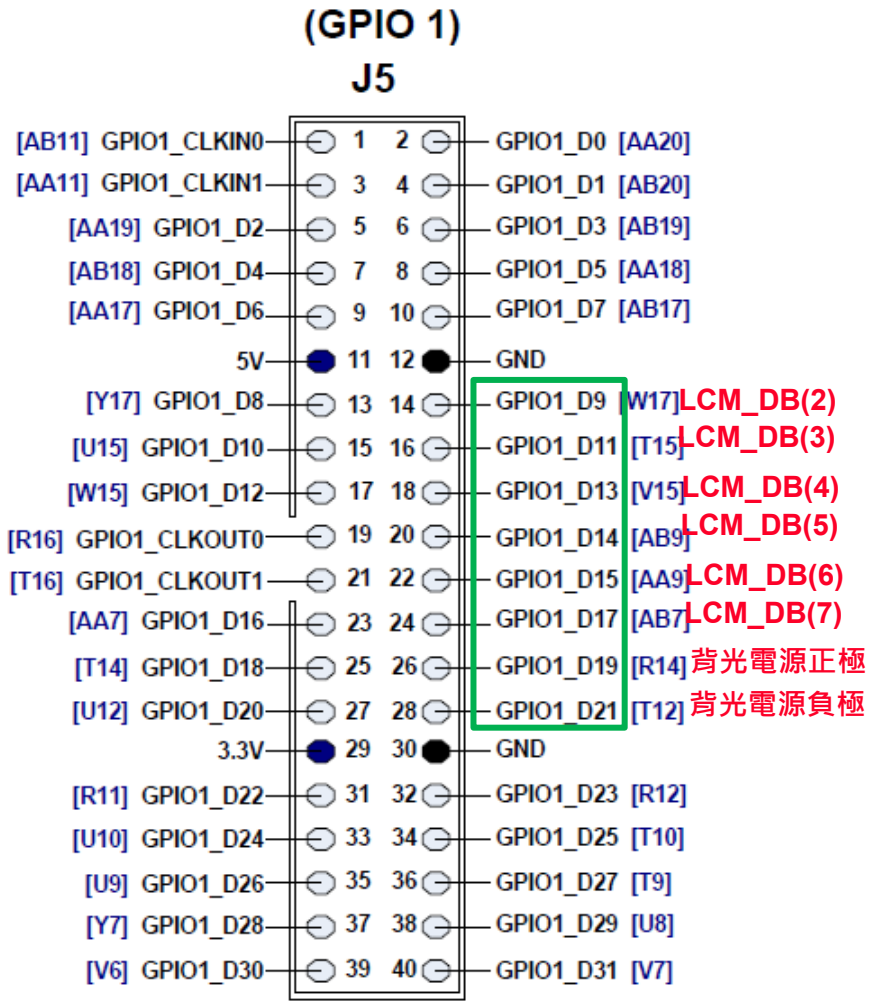
```
ENTITY LCD_Display IS
PORT (  CLOCK_50:in std_logic;
        KEY:in std_logic_vector(2 downto 0);
        GPIO_0:out std_logic_vector(21 downto 9);    -- connect to lcd pin8 to pin1
        GPIO_1:out std_logic_vector(21 downto 9) ); -- connect to lcd pin16 to pin9
END LCD_Display;
```

# Connection between LCD module and DE0 GPIOs



# Connection between LCD module and DE0 GPIOs



# Example 2: Shift Characters on LCD ( lcd_shift.vhd )

- **功能說明：**

使用 CPLD 晶片，設計按鍵控制 LCM 顯示字串移位變化的數位電路。按下 RESET 鍵時，LCM 顯示如圖 12-22 所示靜止畫面；按下 SW0 鍵時，顯示圖 12-22 所示畫面且第 2 列字串每秒右移 1 個字元；按下 SW1 鍵時，顯示圖 12-22 所示畫面且第 2 列字串每秒左移 1 個字元。

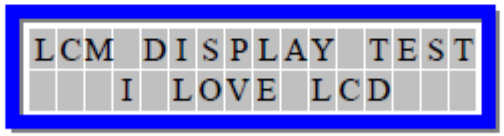



圖12-22　按鍵控制 LCM 顯示字串移位變化的畫面

- Press Key(0) : reset
- Press Key(1) : shift right
- Using Key(2) : shift left

Source: CPLD 數位邏輯設計實務

# Example 2: Shift Characters on LCD ( lcd_shift.vhd )

- 電路方塊圖：

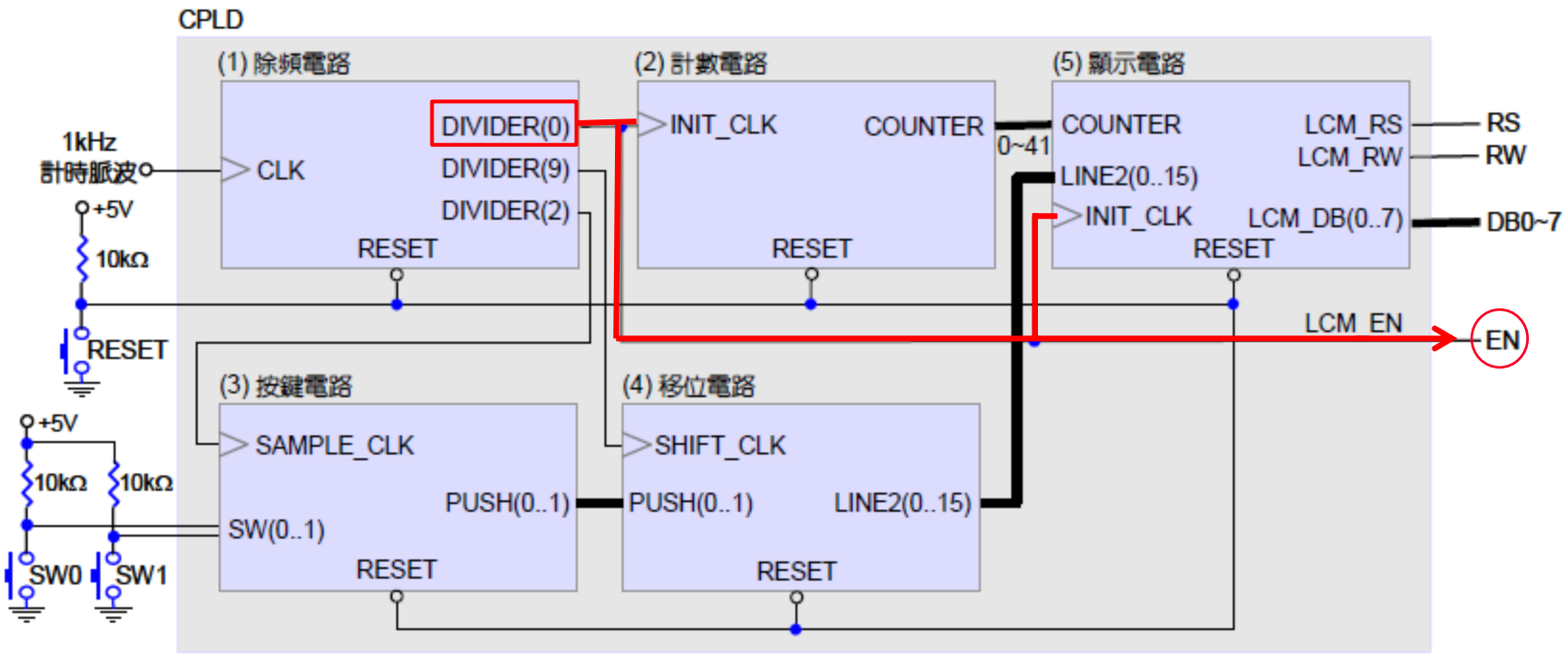


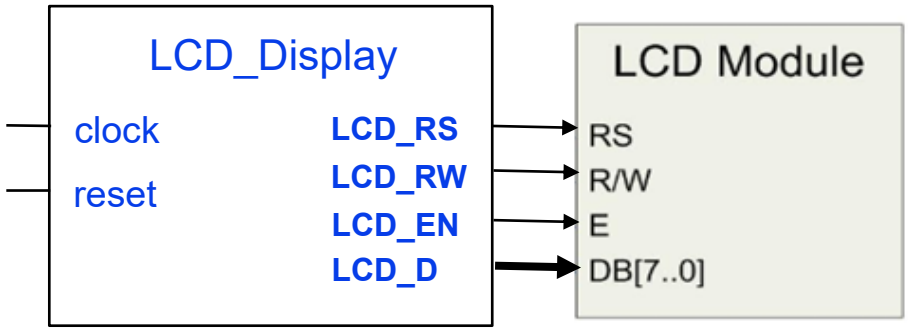

圖12-23　按鍵控制 LCM 顯示字串移位變化電路方塊圖

Source: CPLD 數位邏輯設計實務

# Example 3: Print Strings on LCD ( lcd_display_v1.vhd )

## ➢ Using a Finite State Machine

◆ Print "MCU Digital….." on the 1st line, and "System Design..." on the 2nd line.




```
ENTITY LCD_Display IS
PORT ( clock, reset: IN STD_LOGIC;
       LCD_RS, LCD_RW, LCD_EN: OUT STD_LOGIC;
       LCD_D: OUT STD_LOGIC_VECTOR(7 DOWNTO 0) );
END LCD_Display;
```

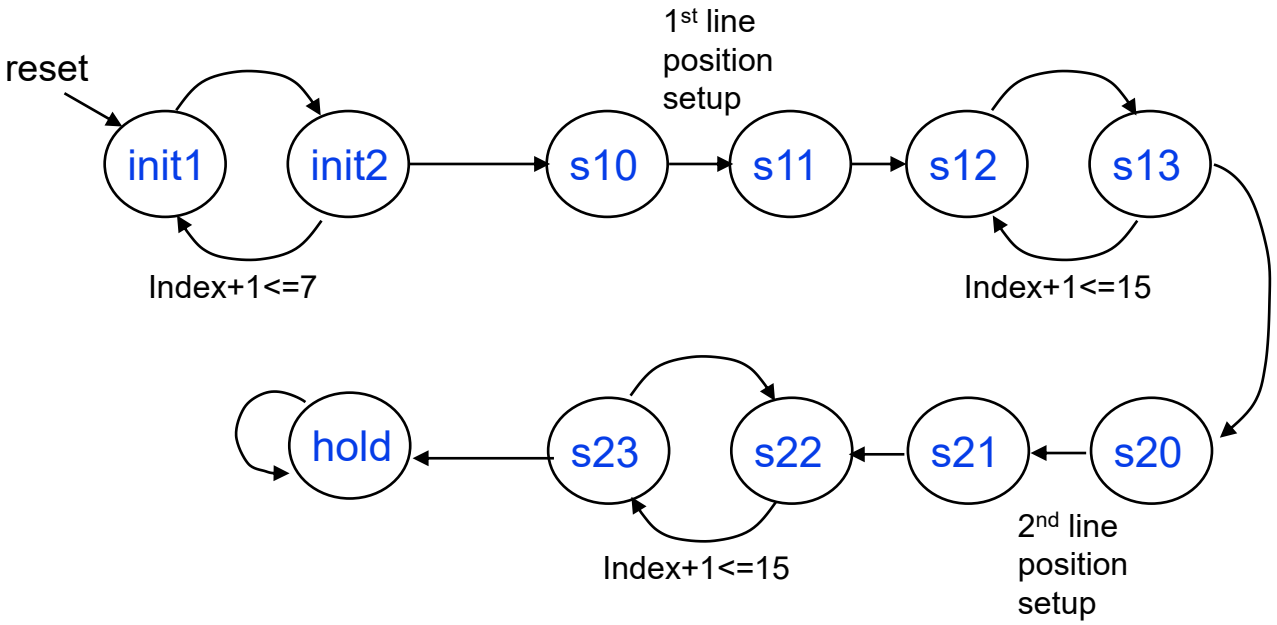
Implementation on DE0

```
ENTITY LCD_Display IS
PORT (  CLOCK_50:in std_logic;
        KEY:in std_logic_vector(2 downto 0);
        GPIO_0:out std_logic_vector(21 downto 9);    -- connect to lcd pin8 to pin1
        GPIO_1:out std_logic_vector(21 downto 9) ); -- connect to lcd pin16 to pin9
END LCD_Display;
```

# Example 3: Print Strings on LCD ( lcd_display_v1.vhd )

## ➢ Using a Finite State Machine

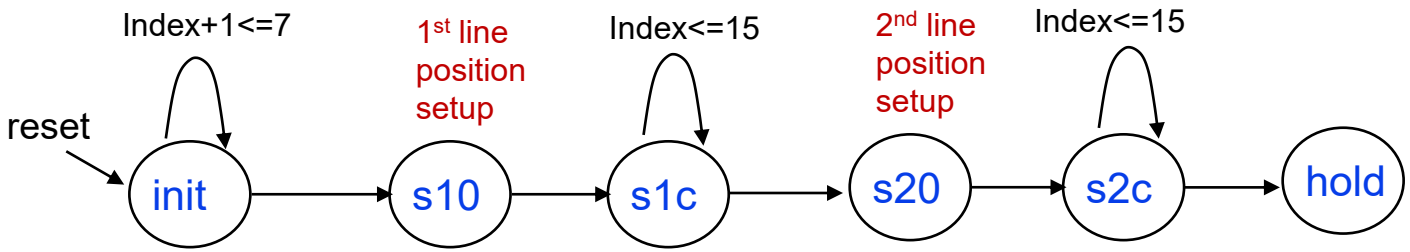
### ◆ State diagram



# Example 4: Print Strings on LCD ( lcd_display_v2.vhd )

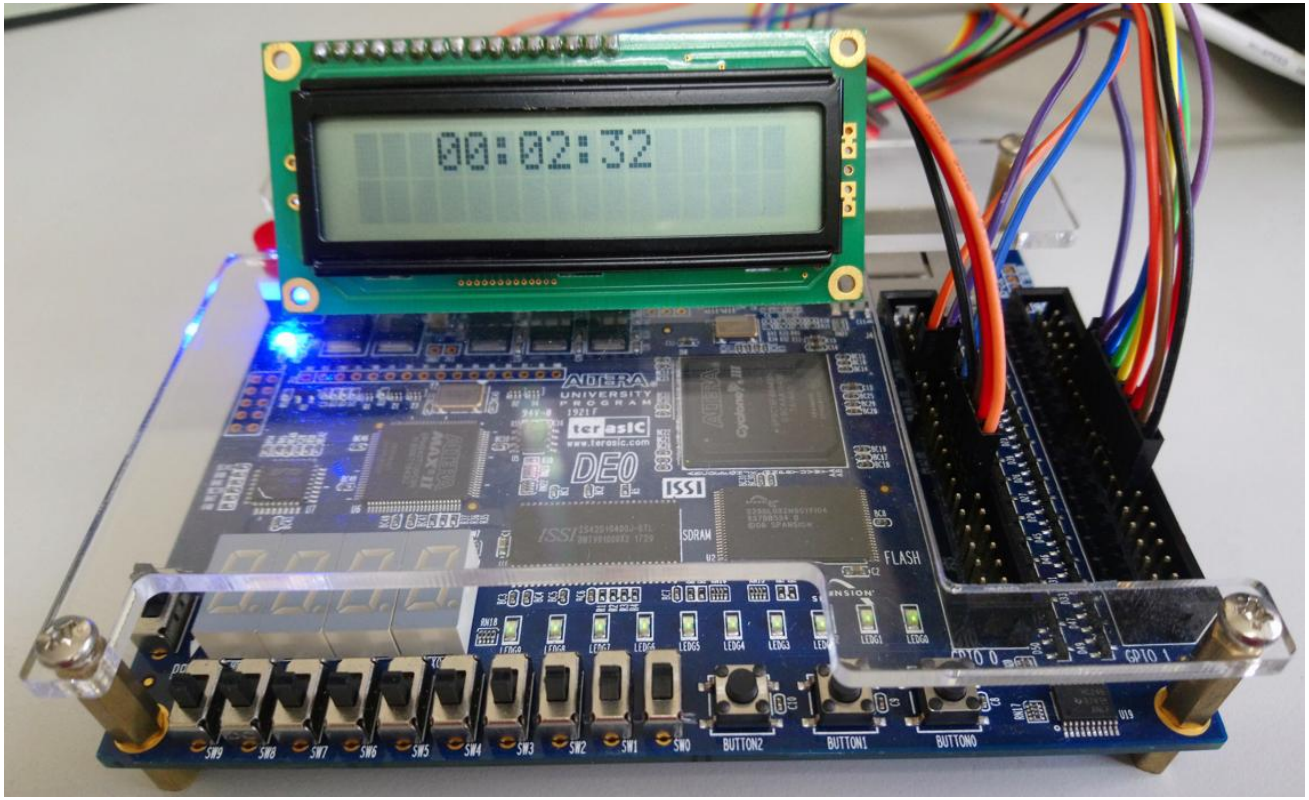
## ➢ Using a Finite State Machine

### ◆ State diagram



# Example 5: 24時制電子鐘 on LCD ( lcd_clock.vhd )

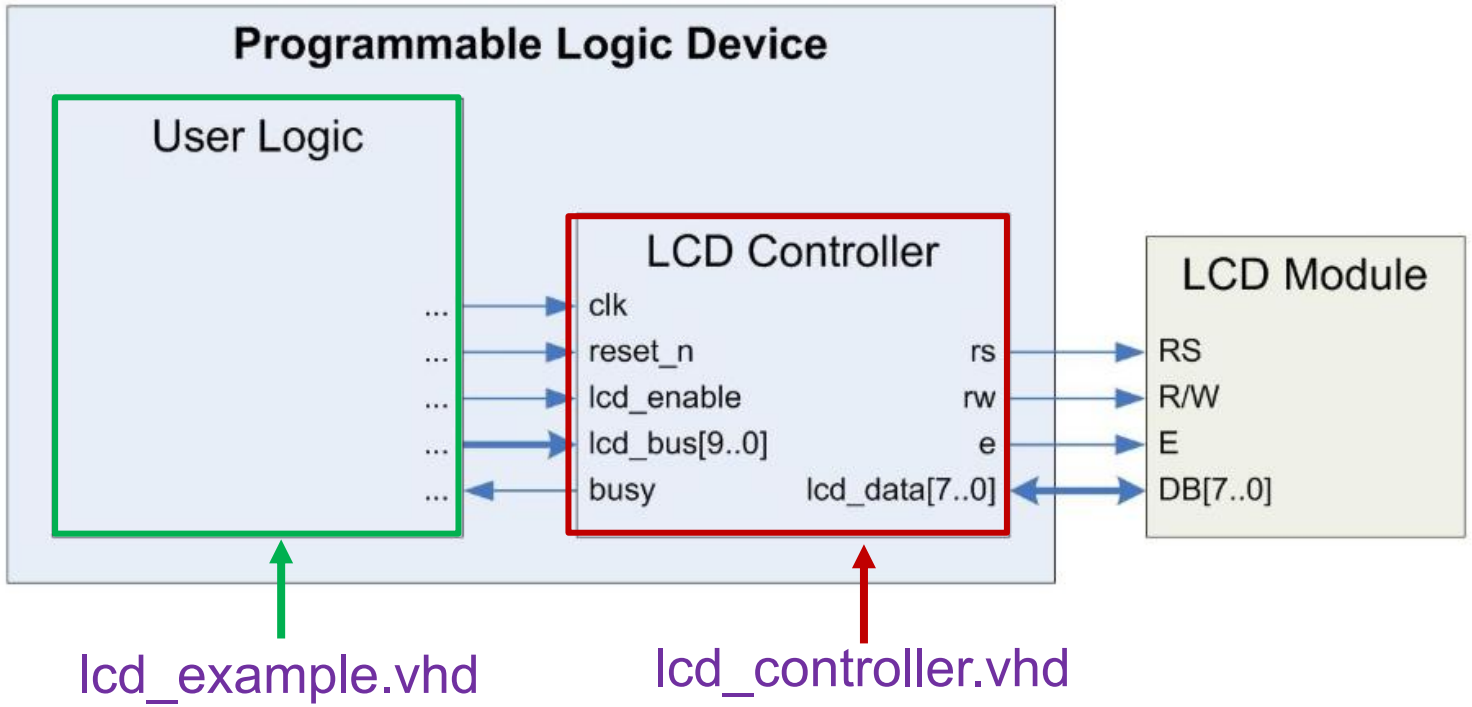
- Show “ HH:MM:SS ” on the 1st line.



# Example 6: Print Strings on LCD

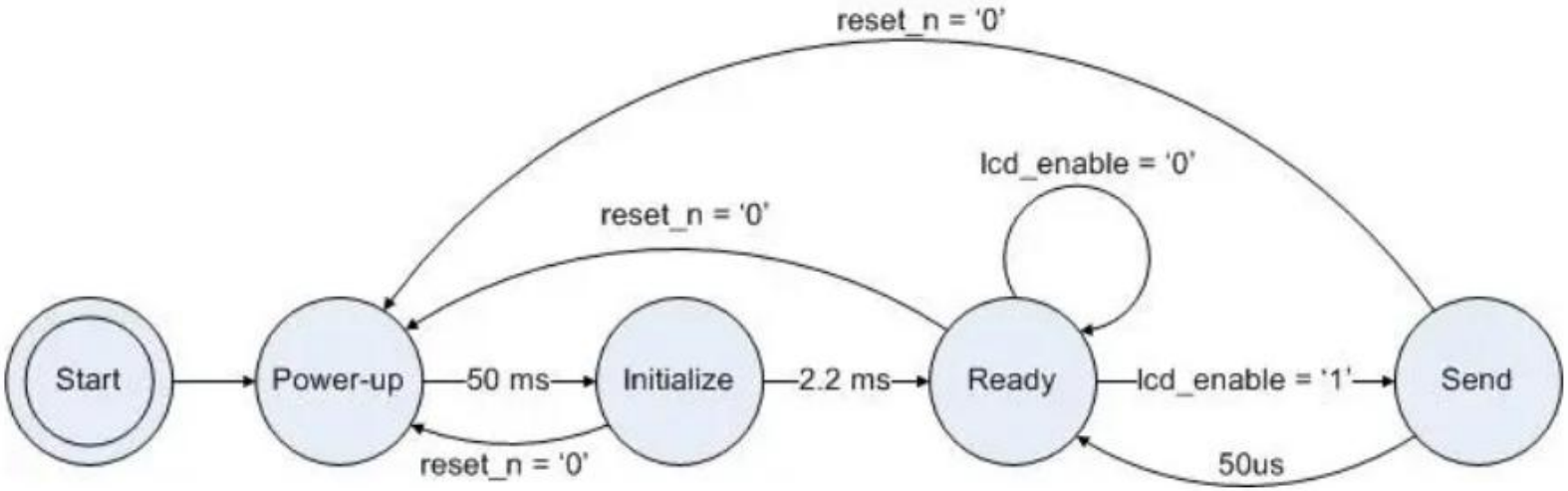
➢ Using a 3rd party IP component (lcd_controller.vhd)

- The controller manages the initialization and data flow to HD44780 compatible 8-bit interface character LCD modules.
- This component allows simple LCD integration into practically any programmable logic application.



# LCD Controller State Machine

- The LCD controller state machine consists of five states. Upon startup, it immediately enters the Power-up state, where it waits 50ms to ensure the supply voltage has stabilized. It then proceeds to an Initialize state.
- The controller cycles the LCD through its initialization sequence, setting the LCD's parameters to default values defined in the hardware. This process completes in approximately 2.2ms, and the controller subsequently assumes a Ready state.
- It waits in this state until the lcd_enable input is asserted, then advances to the Send state. Here, it communicates the appropriate information to the LCD, as defined by the lcd_bus input. After 50us, it returns to the Ready state until further notice.
- If a low logic level is applied to the reset_n input at any time for a minimum of one clock cycle, the controller resets to the Power-up state and re-initializes.



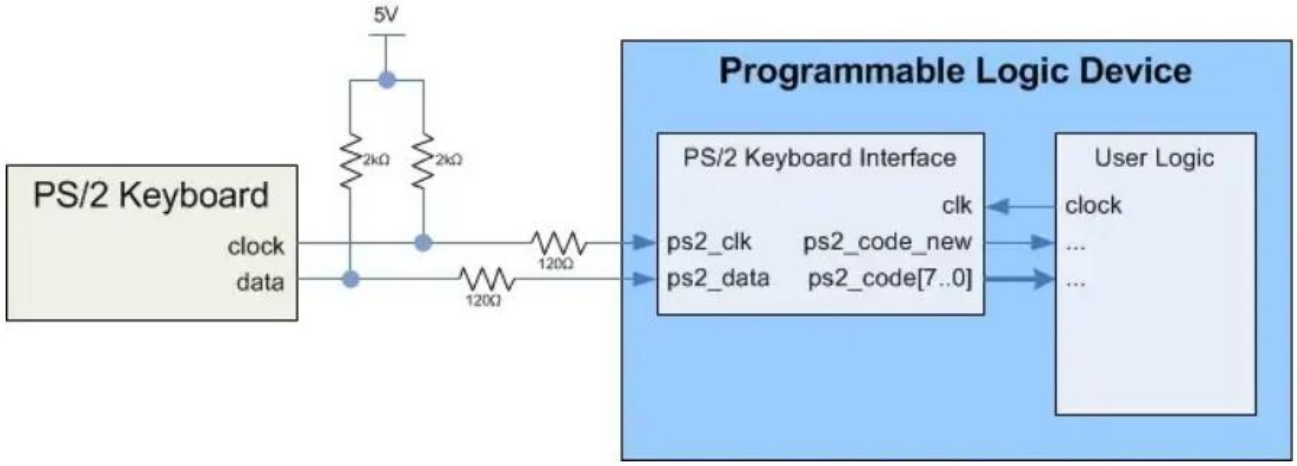
# LCD Controller State Machine

- The LCD controller executes an initialization sequence each time it is powered-up or the reset_n pin is deasserted for a minimum of one clock cycle. The controller asserts the busy pin during initialization.
- The initialization sequence specifies several LCD parameters: function, display control, display clear, and entry mode. The LCD controller instantiates the following default set of these options.
  - Function Set: 2-line mode, display on
  - Display Control: display on, cursor off, blink off
  - Entry Mode: increment mode, entire shift off
- The user can send commands to the LCD to change any parameters after initialization.

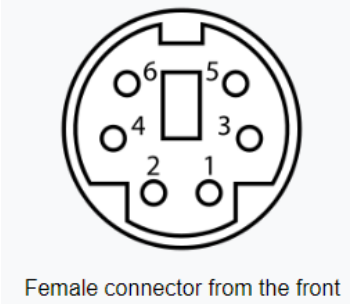
# PS/2 Keyboard

# PS/2 Keyboard Interface

- The PS/2 interface was originally developed for the IBM PC/AT's mouse and keyboard in 1984. The Altera FPGA boards support the use of either a mouse or keyboard using a PS/2 connector on the board (not both at the same time).




- The PS/2 port consists of 6 pins including ground, power (VDD), keyboard data, and a keyboard clock line.

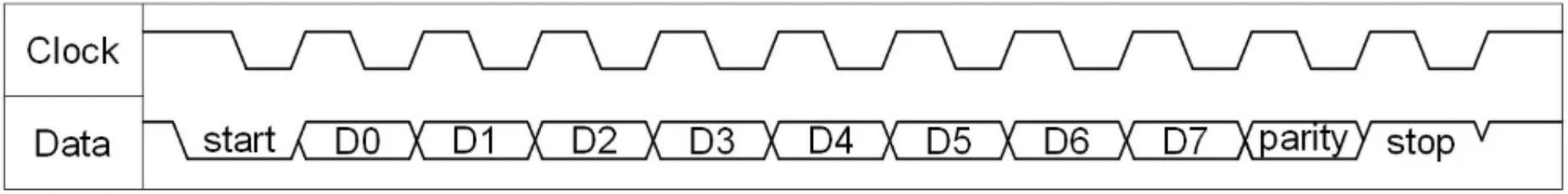


Female connector from the front

| | | |
|---|---|---|
| **Pin 1** | +DATA | Data |
| **Pin 2** | | Not connected[b] |
| **Pin 3** | GND | Ground |
| **Pin 4** | Vcc | +5 V DC at 275 mA |
| **Pin 5** | +CLK | Clock |
| **Pin 6** | | Not connected[c] |

# PS/2 Keyboard Transmission Timing Diagram

- The keyboard provides both the clock and data. The clock has a frequency between 10 kHz and 16.7 kHz (i.e. a 60-100us period).
- The data begins with a start bit (logic low), followed by one byte of data, a parity bit, and finally a stop bit (logic high).
- The data is sent LSB first. Each bit should be read on the falling edge of the clock signal.
- Once complete, both the clock and data signals return to logic level high.

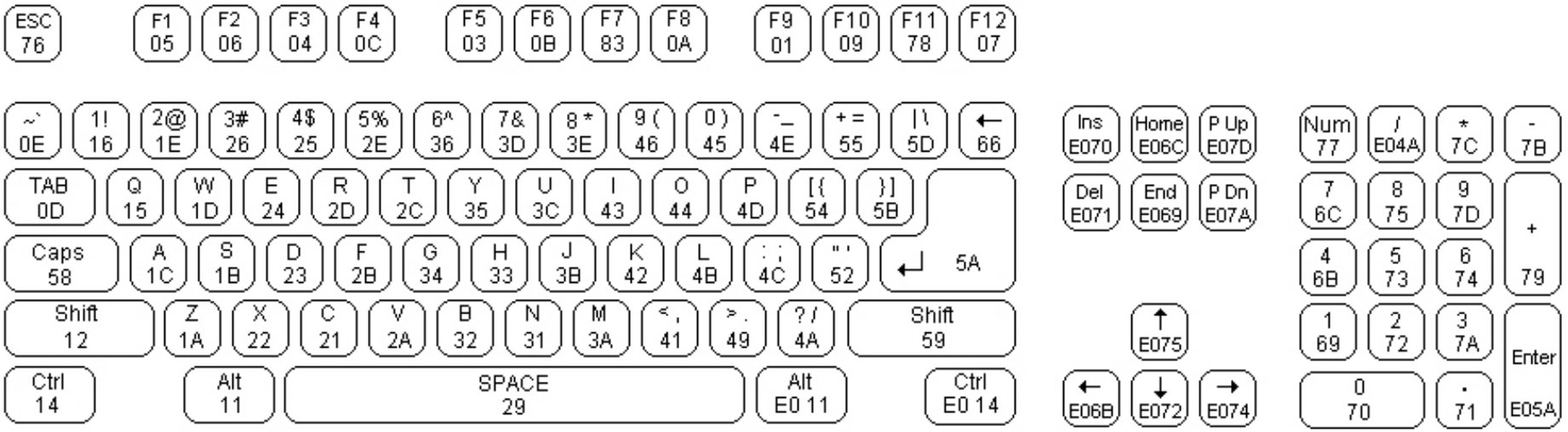



1. A start bit ('0')
2. 8 data bits containing the key scan code in low to high bit order
3. Odd parity bit such that the eight data bits plus the parity bit are an odd number of ones
4. A stop bit ('1')

# Scan Codes

- The data byte represents part of a keyboard scan code: either a make code (key press) or a break code (key release).
- One make code is sent every time a key is pressed. When a key is released, a break code is sent.
- A make code usually consists of either one or two bytes. If a make code uses two bytes, the first byte is x“E0”.
- A given key’s break code is typically the same as its make code, except that break codes include an additional x“F0” byte as the 2nd to last byte.

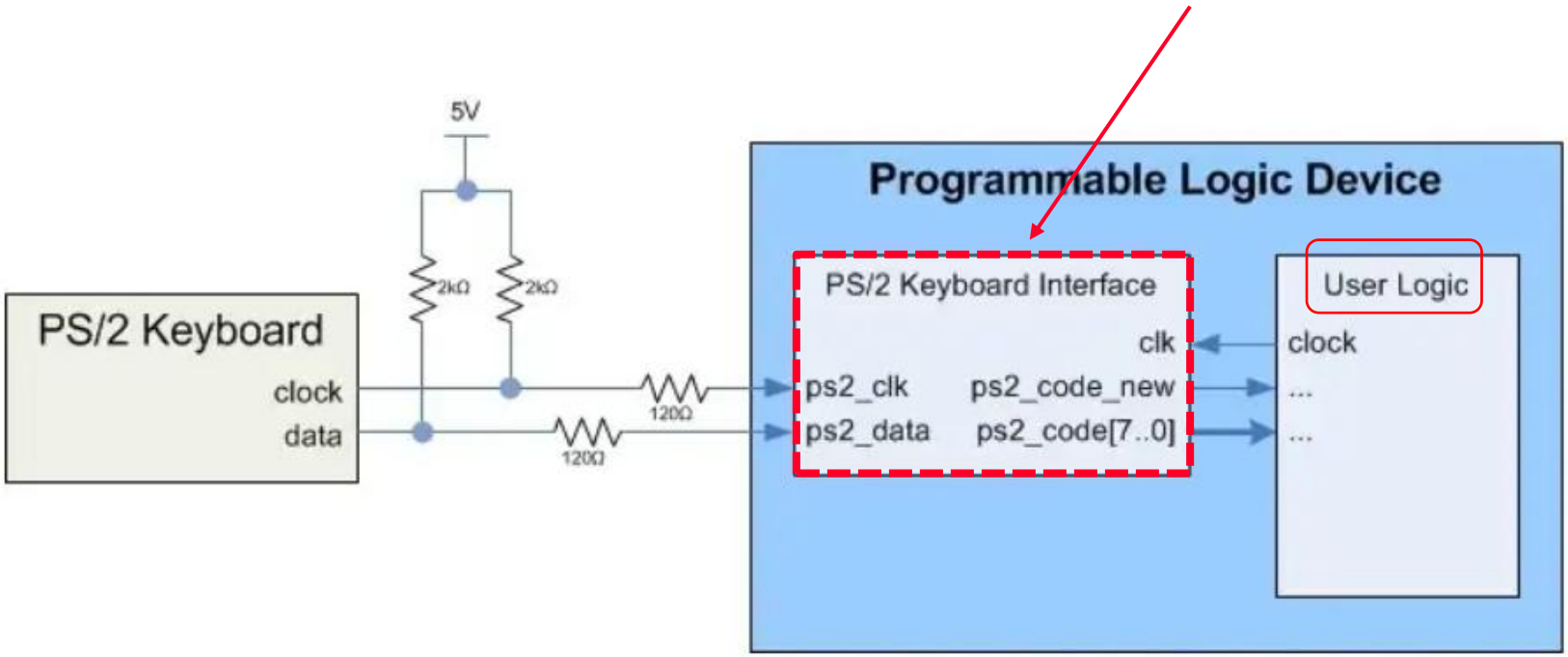
# Common Scan Codes for PS/2 Keyboard



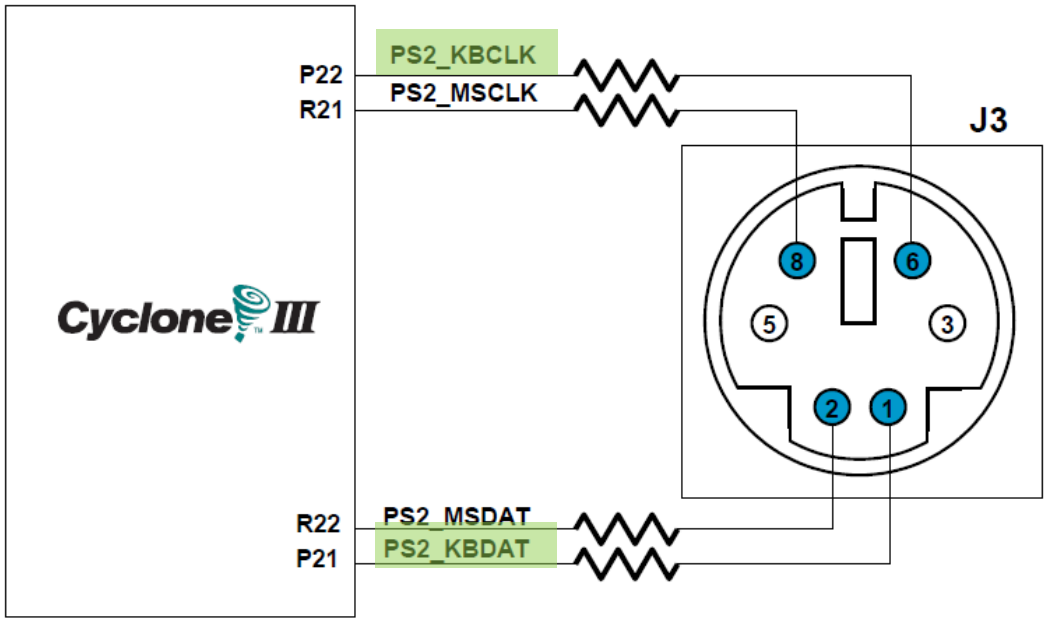
# Common Scan Codes for PS/2 Keyboard

| KEY | MAKE | BREAK | KEY | MAKE | BREAK | KEY | MAKE | BREAK |
|---|---|---|---|---|---|---|---|---|
| A | 1C | F0,1C | 0 | 45 | F0,45 | [ | 54 | FO,54 |
| B | 32 | F0,32 | 1 | 16 | F0,16 | INSERT | E0,70 | E0,F0,70 |
| C | 21 | F0,21 | 2 | 1E | F0,1E | HOME | E0,6C | E0,F0,6C |
| D | 23 | F0,23 | 3 | 26 | F0,26 | PG UP | E0,7D | E0,F0,7D |
| E | 24 | F0,24 | 4 | 25 | F0,25 | DELETE | E0,71 | E0,F0,71 |
| F | 2B | F0,2B | 5 | 2E | F0,2E | END | E0,69 | E0,F0,69 |
| G | 34 | F0,34 | 6 | 36 | F0,36 | PG DN | E0,7A | E0,F0,7A |
| H | 33 | F0,33 | 7 | 3E | F0,3D | U ARROW | E0,75 | E0,F0,75 |
| I | 43 | F0,43 | 8 | 58 | F0,3E | L ARROW | E0,6B | E0,F0,6B |
| J | 3B | F0,3B | 9 | 46 | F0,46 | D ARROW | E0,72 | E0,F0,72 |
| K | 42 | F0,42 | ` | 0E | F0,0E | R ARROW | E0,74 | E0,F0,74 |
| L | 4B | F0,4B | - | 4E | F0,4E | NUM | 77 | F0,77 |
| M | 3A | F0,3A | = | 55 | FO,55 | ENTER | 5A | F0,5A |
| N | 31 | F0,31 | \ | 5D | F0,5D | ESC | 76 | F0,76 |
| O | 44 | F0,44 | BKSP | 66 | F0,66 | ] | 5B | F0,5B |
| P | 4D | F0,4D | SPACE | 29 | F0,29 | ; | 4C | F0,4C |
| Q | 15 | F0,15 | TAB | 0D | F0,0D | ' | 52 | F0,52 |
| R | 2D | F0,2D | CAPS | 58 | F0,58 | , | 41 | F0,41 |
| S | 1B | F0,1B | L SHFT | 12 | FO,12 | . | 49 | F0,49 |
| T | 2C | F0,2C | L CTRL | 14 | FO,14 | / | 4A | F0,4A |
| U | 3C | F0,3C | L WIN | E0,1F | E0,F0,1F | | | |
| V | 2A | F0,2A | L ALT | 11 | F0,11 | | | |
| W | 1D | F0,1D | R SHFT | 59 | F0,59 | | | |
| X | 22 | F0,22 | R CTRL | E0,14 | E0,F0,14 | | | |
| Y | 35 | F0,35 | R WIN | E0,27 | E0,F0,27 | | | |
| Z | 1A | F0,1A | R ALT | E0,11 | E0,F0,11 | | | |

# VHDL for PS/2 Keyboard Interface ( ps2_keyboard.vhd )



# Connection between PS/2 Serial Port and DE0




KB: Keyboard

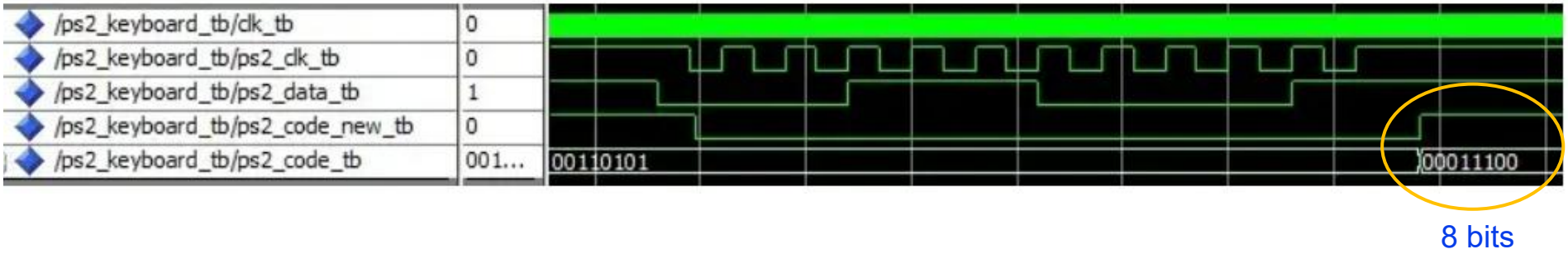
MS: Mouse

Table 4.13. PS/2 pin assignments

| Signal Name | FPGA Pin No. | Description |
|---|---|---|
| PS2_KBCLK | PIN_P22 | PS/2 Clock |
| PS2_KBDAT | PIN_P21 | PS/2 Data |
| PS2_MSCLK | PIN_R21 | PS/2 Clock (reserved for second PS/2 device) |
| PS2_MSDAT | PIN_R22 | PS/2 Data(reserved for second PS/2 device) |

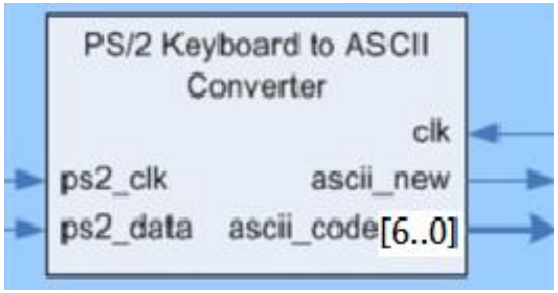
# Transaction

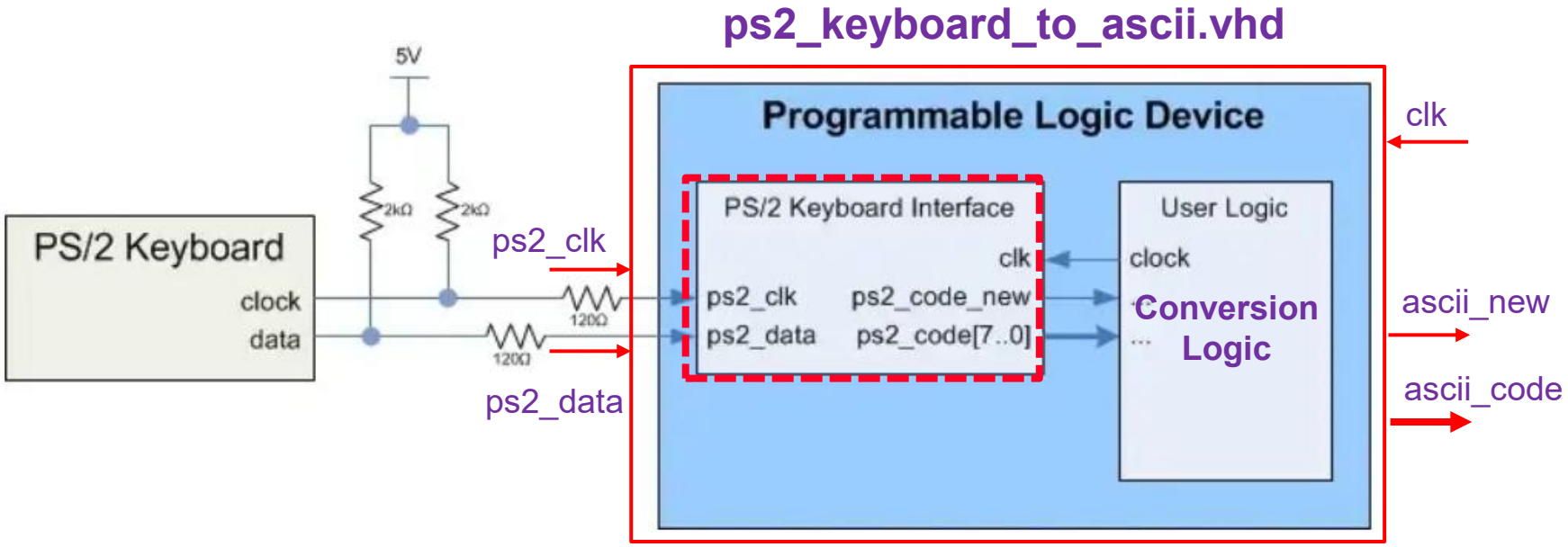
- Once the PS/2 clock signal goes low, the ps2_code_new flag deasserts to indicate that a new PS/2 keyboard transaction is in progress.
- When the transaction completes, the ps2_code_new flag asserts to indicate that a new PS/2 code has been received and is available on the ps2_code bus.
- In this case, the PS/2 code received is x“1C”, which is the make code for the “A” key.



# Example: PS/2 Keyboard to ASCII Converter

- The component receives data transactions from a PS/2 keyboard and provides the corresponding ASCII codes to user logic over a parallel interface.



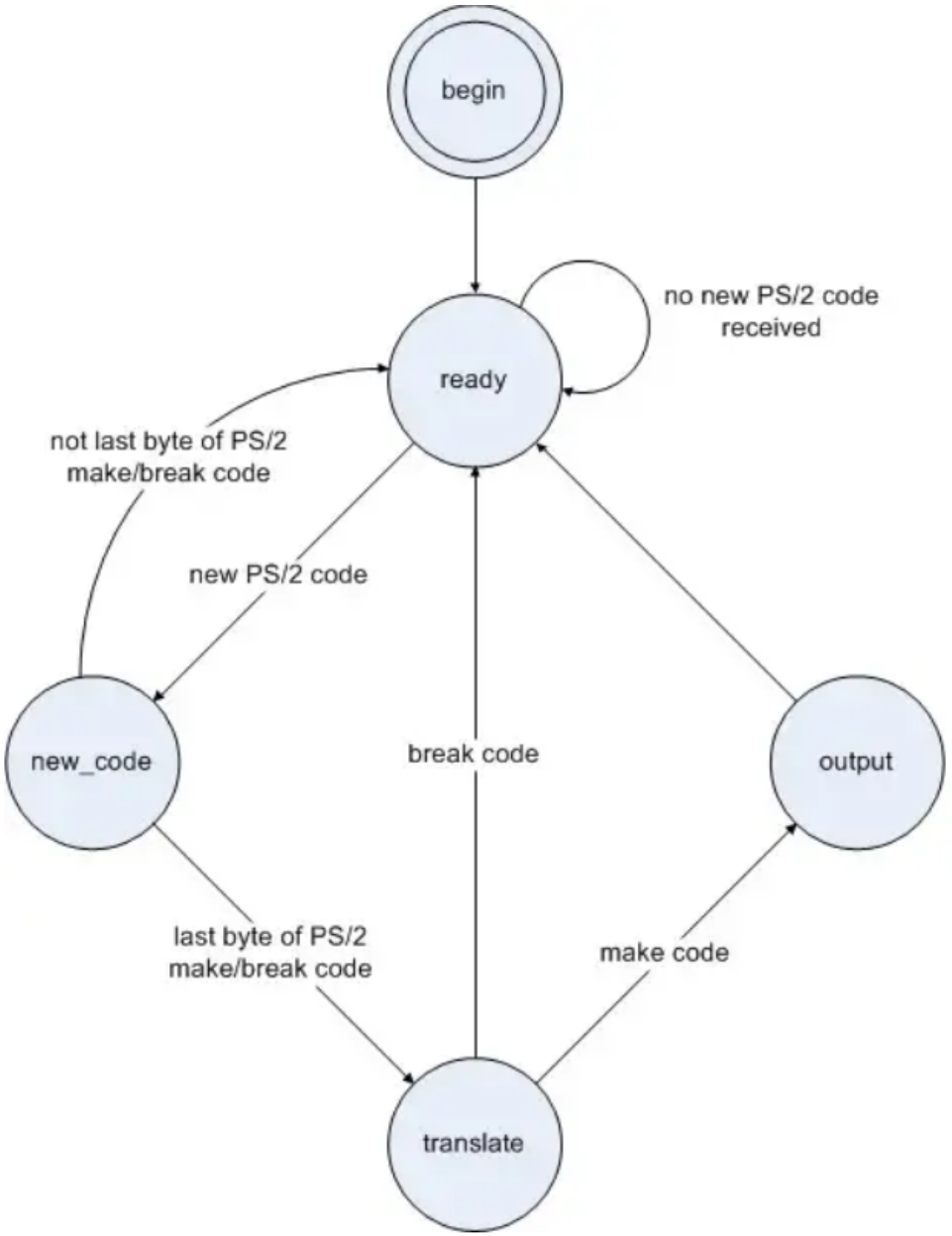



# Example: PS/2 Keyboard to ASCII Converter

- The PS/2 codes provided by the PS/2 keyboard interface component control the converter's state machine.
  - Upon start-up, the component immediately enters the ready state. It waits in this state until it receives a new PS/2 code.
  - The new_code state builds the PS/2 make or break codes. If the new code received is the last byte in the make/break code, the state machine proceeds to the translate state, otherwise it returns to the ready state to await the next byte.
  - Once in the translate state, the converter determines which key was pressed and translates it into ASCII.
    - If the code is a break code, then no action is needed, so the converter ignores the code and returns to the ready state.
    - However, if a make code was received, the converter proceeds to the output state, where it outputs the resultant ASCII code on the ascii_code bus and sets the ascii_new output flag to indicate that the new code is available.
  - The converter then returns to the ready state to await the next communication from the PS/2 keyboard interface component.

# Example: PS/2 Keyboard to ASCII Converter

- State Diagram:



# VHDL for PS/2 Keyboard to ASCII Converter

```
--new_code state: determine what to do with the new PS2 code
WHEN new_code =>
  IF(ps2_code = x"F0") THEN     --code indicates that next command is break
    break <= '1';                 --set break flag
    state <= ready;               --return to ready state to await next PS2 code
  ELSIF(ps2_code = x"E0") THEN --code indicates multi-key command
    e0_code <= '1';               --set multi-code command flag
    state <= ready;               --return to ready state to await next PS2 code
  ELSE                           --code is the last PS2 code in the make/break code
    ascii(7) <= '1';              --set internal ascii value to unsupported code
    state <= translate;           --proceed to translate state      (for verification)
  END IF;

--translate state: translate PS2 code to ASCII value
WHEN translate =>
    break <= '0';     --reset break flag
    e0_code <= '0';   --reset multi-code command flag

    --handle codes for control, shift, and caps lock
    CASE ps2_code IS
      WHEN x"58" =>                    --caps lock code
        IF(break = '0') THEN             --if make command
          caps_lock <= NOT caps_lock;      --toggle caps lock
        END IF;
      WHEN x"14" =>                    --code for the control keys
        IF(e0_code = '1') THEN           --code for right control
          control_r <= NOT break;          --update right control flag
        ELSE                             --code for left control
          control_l <= NOT break;          --update left control flag
        END IF;
      WHEN x"12" =>                    --left shift code
        shift_l <= NOT break;            --update left shift flag
      WHEN x"59" =>                    --right shift code
        shift_r <= NOT break;            --update right shift flag
      WHEN OTHERS => NULL;
    END CASE;
```

Note: these assignments are completed only after the process ends. So far, they're still the values determined in "new_code" state.

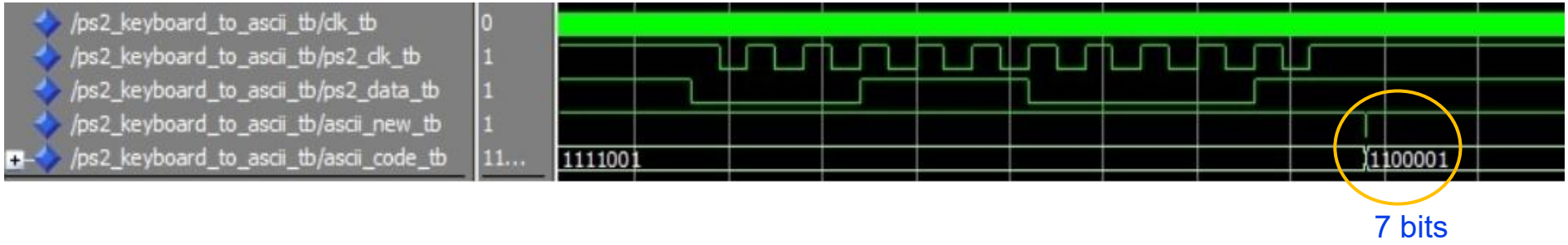
# ASCII Code

**Table 1.7**
*American Standard Code for Information Interchange (ASCII)*

| $b_4b_3b_2b_1$ | $b_7b_6b_5$ = 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 0000 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 1000 | BS | CAN | ( | 8 | H | X | h | x |
| 1001 | HT | EM | ) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [ | k | { |
| 1100 | FF | FS | , | < | L | \ | l | \| |
| 1101 | CR | GS | − | = | M | ] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | − | o | DEL |

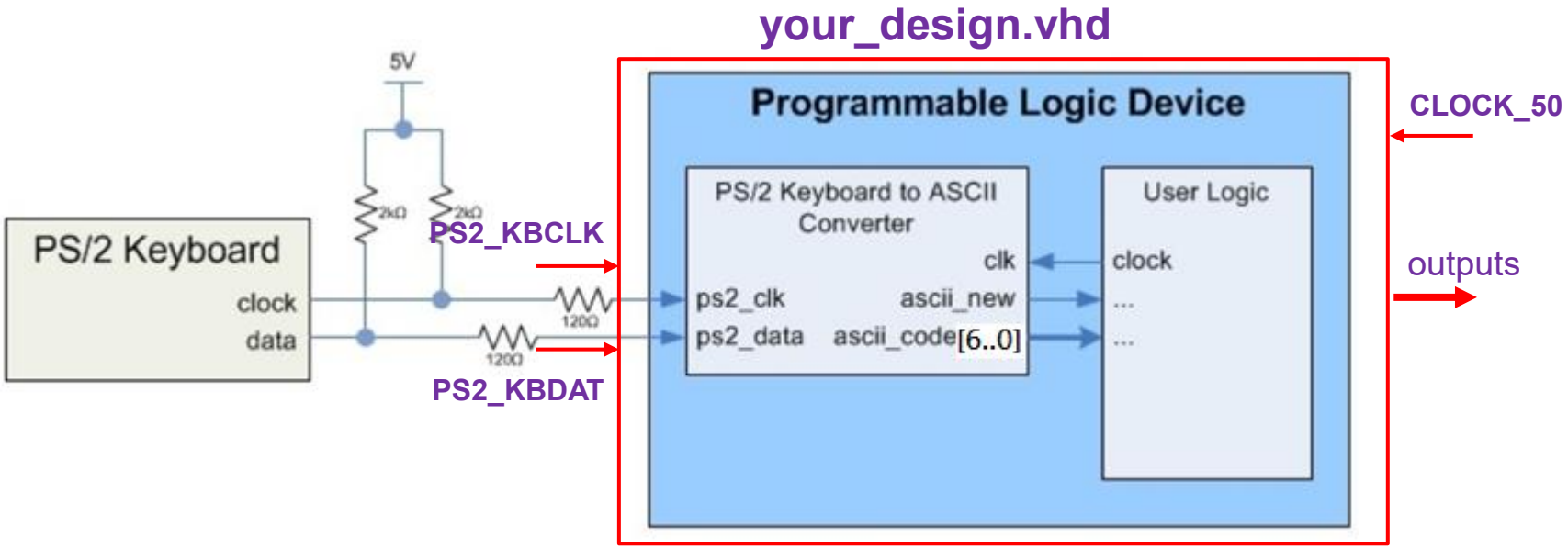
# Transaction

- Once the internal PS/2 keyboard interface finishes receiving the PS/2 transaction, the ascii_new flag deasserts to indicate that a new PS/2 to ASCII conversion is in progress.
- When the transaction completes, the ascii_new flag asserts to indicate that a new ASCII code has been received and is available on the ascii_code bus.
- In this case, the PS/2 code received is x“1C” (the make code for the “A” key), and the resulting ASCII code is x“61” (the ASCII code for the “a” character).

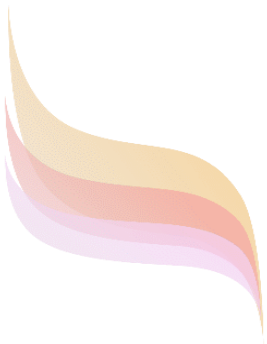


# Application of PS/2 Keyboard to ASCII Converter

- The application (your_design.vhd in DE0) consists of two parts:
  - ps2_keyboard_to_ascii.vhd
  - User Logic
- The ps2_keyboard_to_ascii.vhd instantiates the PS/2 keyboard interface component (ps2_keyboard.vhd). The PS/2 keyboard interface component in turn instantiates the debounce component (debounce.vhd).

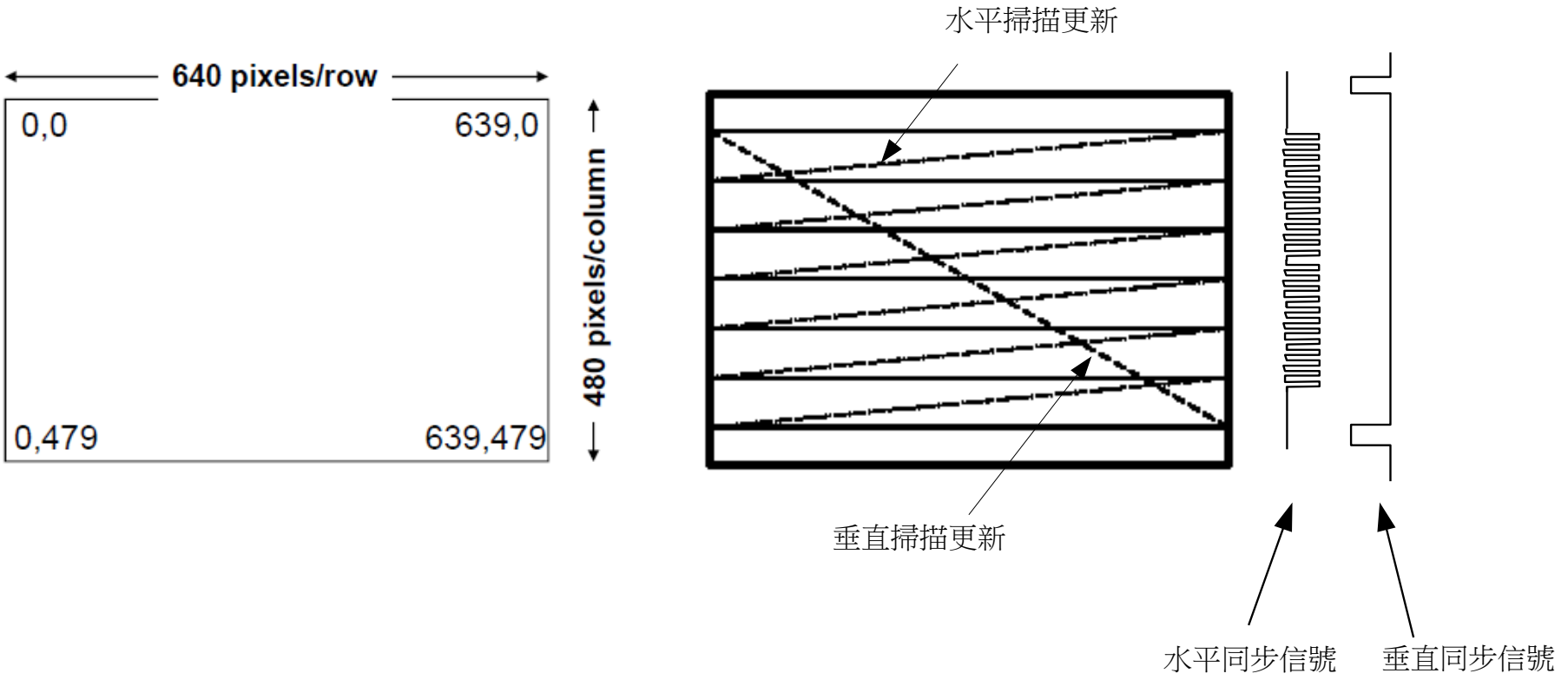


# VGA螢幕顯示



# VGA Video Display Technology

- In standard VGA format, the screen contains 640x480 pixels
  - 640 pixels in a row
  - 480 rows
- The standard refresh rate for a screen is ~ 60 Hz
  - The entire screen is refreshed 60 times per second



# VGA Clock Information

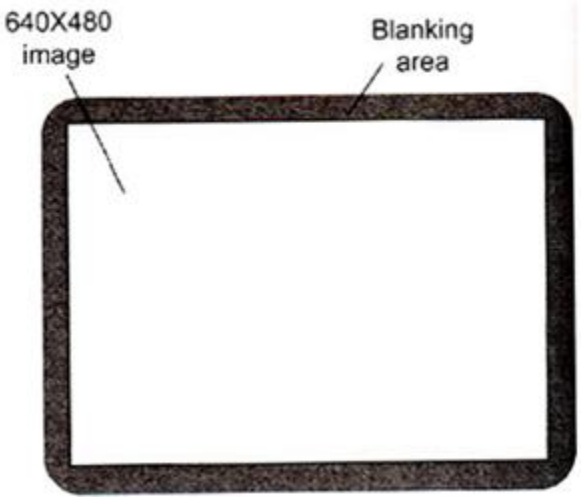
在標準的 VGA 格式中，整個螢幕包含 640×480 個圖素（pixels），而視訊信號固定以每秒 60 次的頻率更新整個螢幕畫面，這個顯示頻率超過人眼視覺暫留所需的 24Hz，因此畫面看起來是穩定而不閃爍的。由於 640×480 解析度的圖框（frame）顯示需要 800（dots）×525（Lines）來掃描完成（包含了遮沒區域），所以我們可以求得每一個點的更新率（Refresh rate）如下：

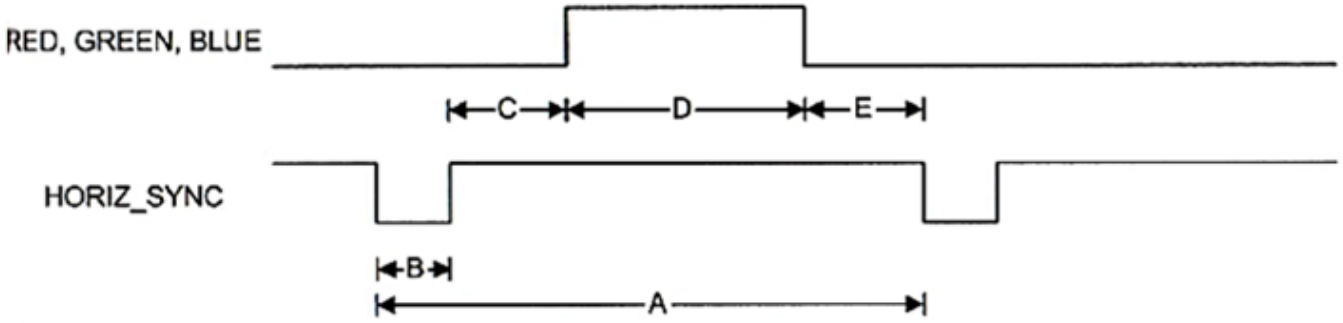
$$1\,s / (800 \times 525 \times 60\ \text{times}) \cong 40\,ns$$

Altera DE0 實驗板已提供 50MHz 之振盪頻率，將該頻率除二之後，便可得到 25MHz 的輸出頻率，且其週期爲 40ns：

$$1 / 25MHz = 40\,ns$$

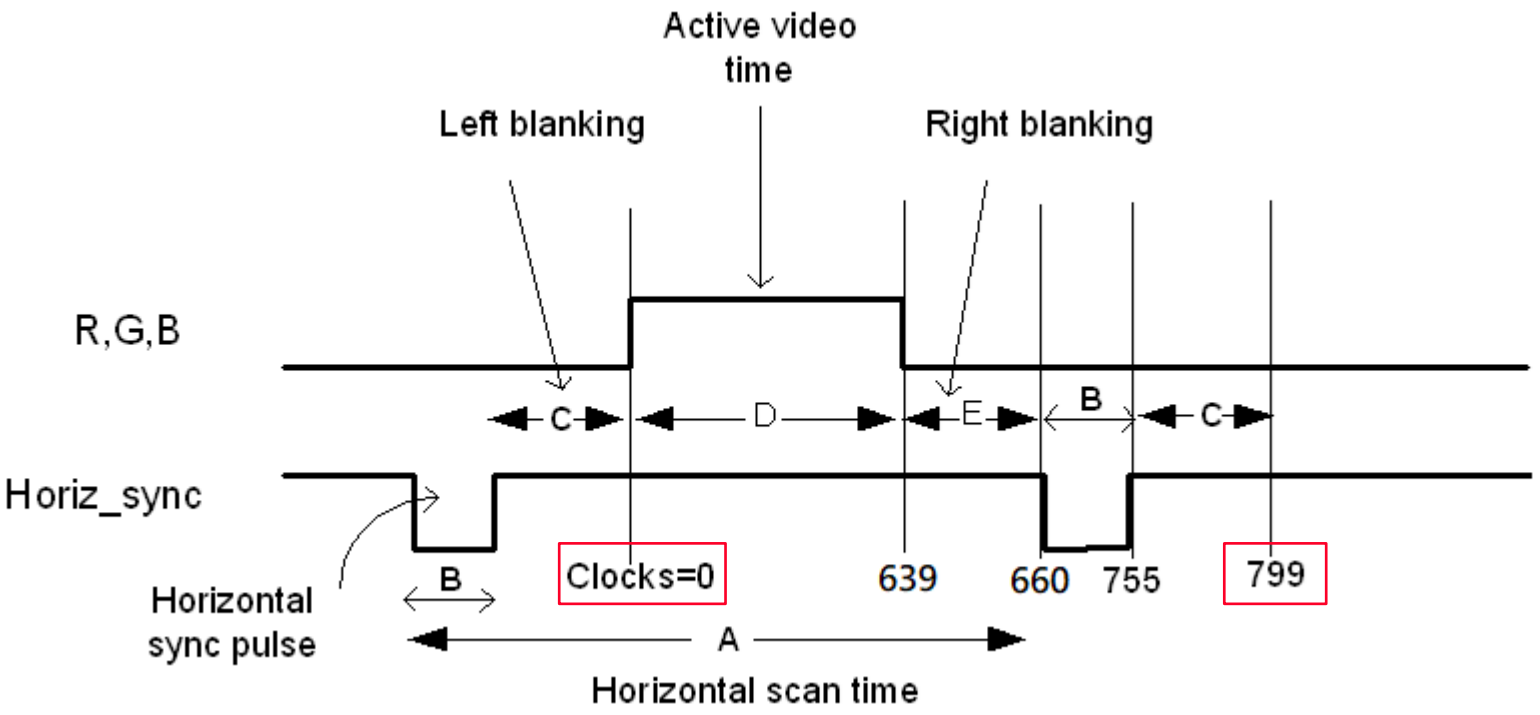
# Horizontal Refresh Cycle



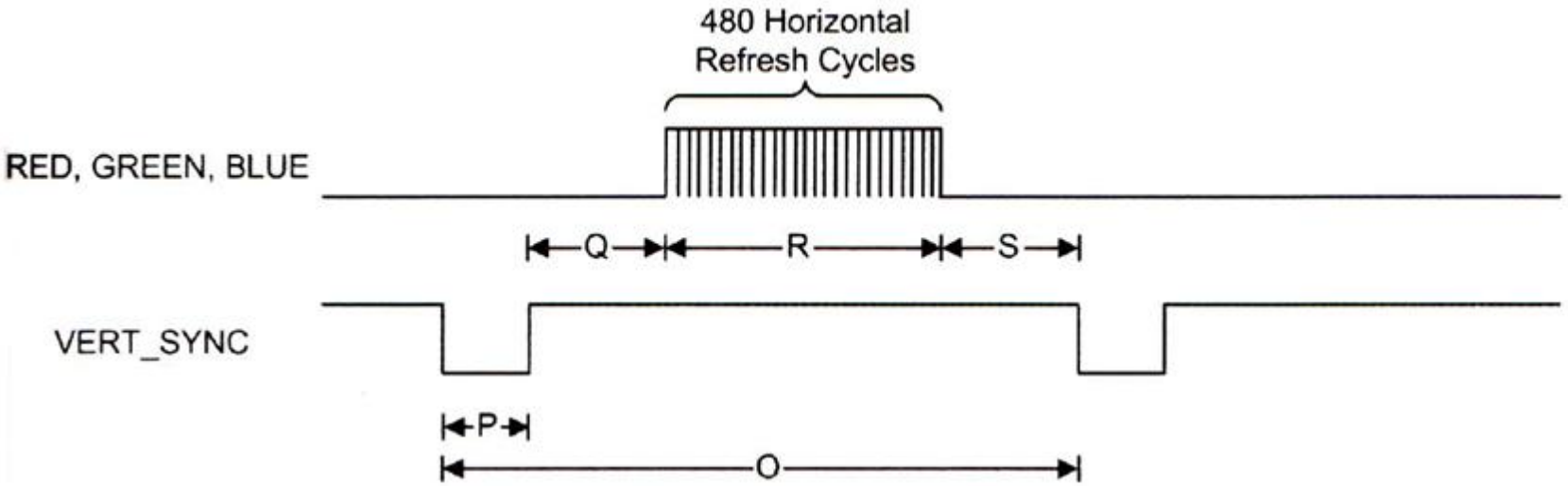

| Parameters | A | B | C | D | E |
|---|---|---|---|---|---|
| Time | 31.77μs | 3.77μs | 1.89μs | 25.17μs | 0.94μs |

# 水平同步信號波形與其對應的計數值



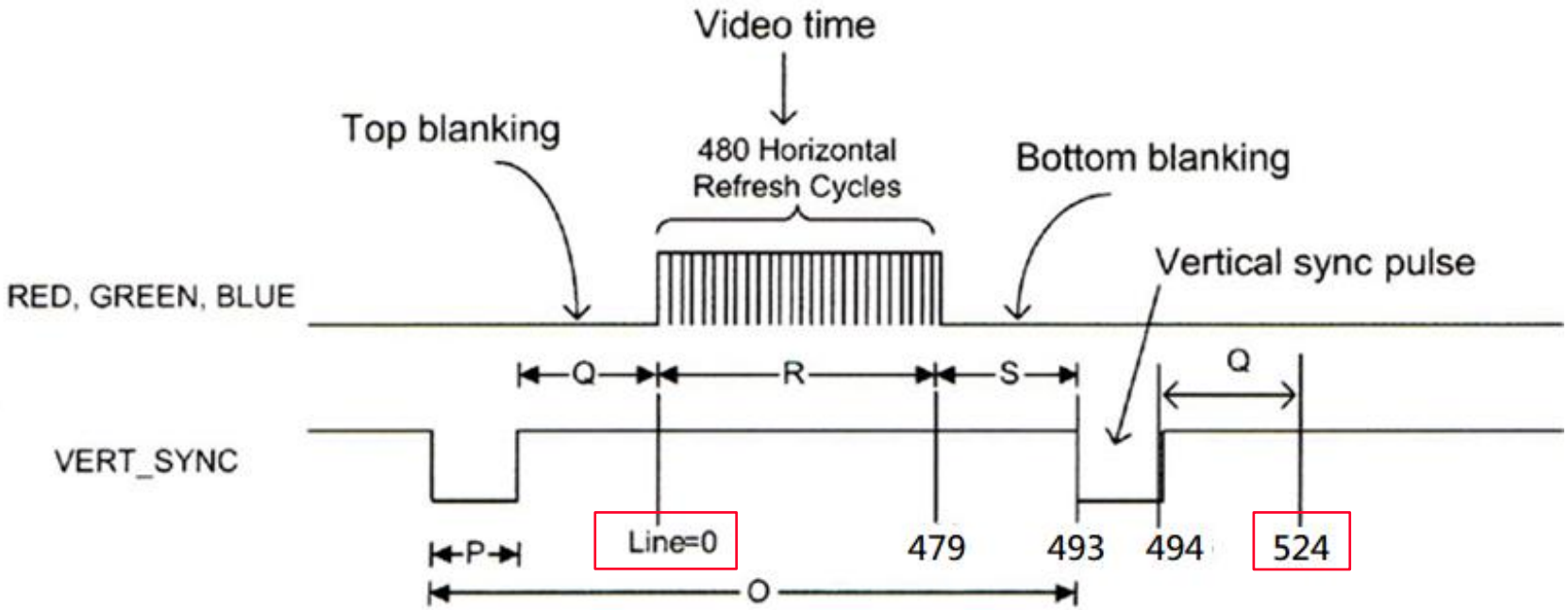

| Parameters | A | B | C | D | E |
|---|---|---|---|---|---|
| Time | 31.77us | 3.77us | 1.89us | 25.17us | 0.94us |
| Clocks | 800 | 96 | 44 | 640 | 20 |

# Vertical Refresh Cycle




| Parameters | O | P | Q | R | S |
|---|---|---|---|---|---|
| Time | 16.6ms | 64μs | 1.02ms | 15.25ms | 0.35ms |

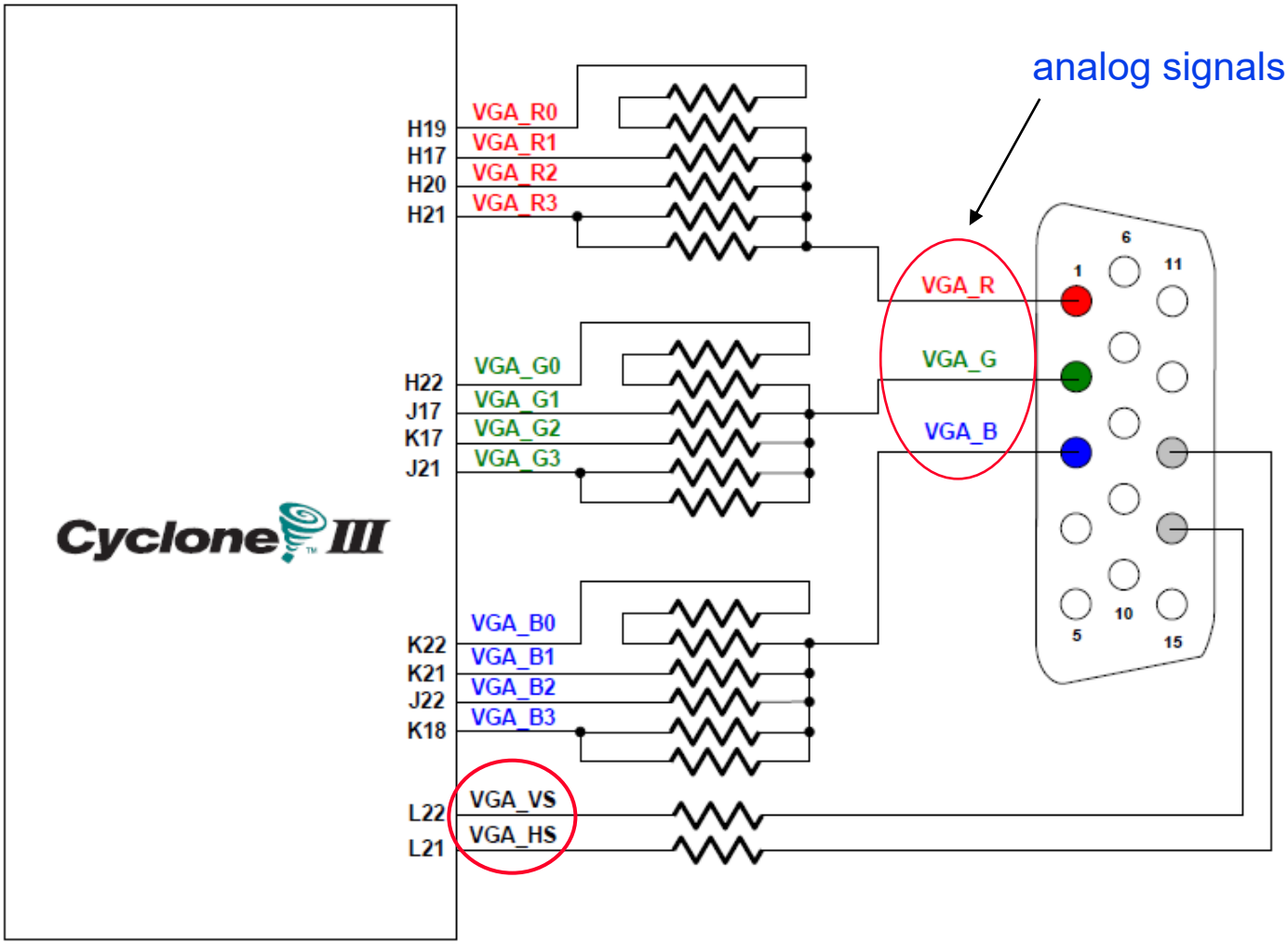
# 垂直同步信號波形與其對應的計數值



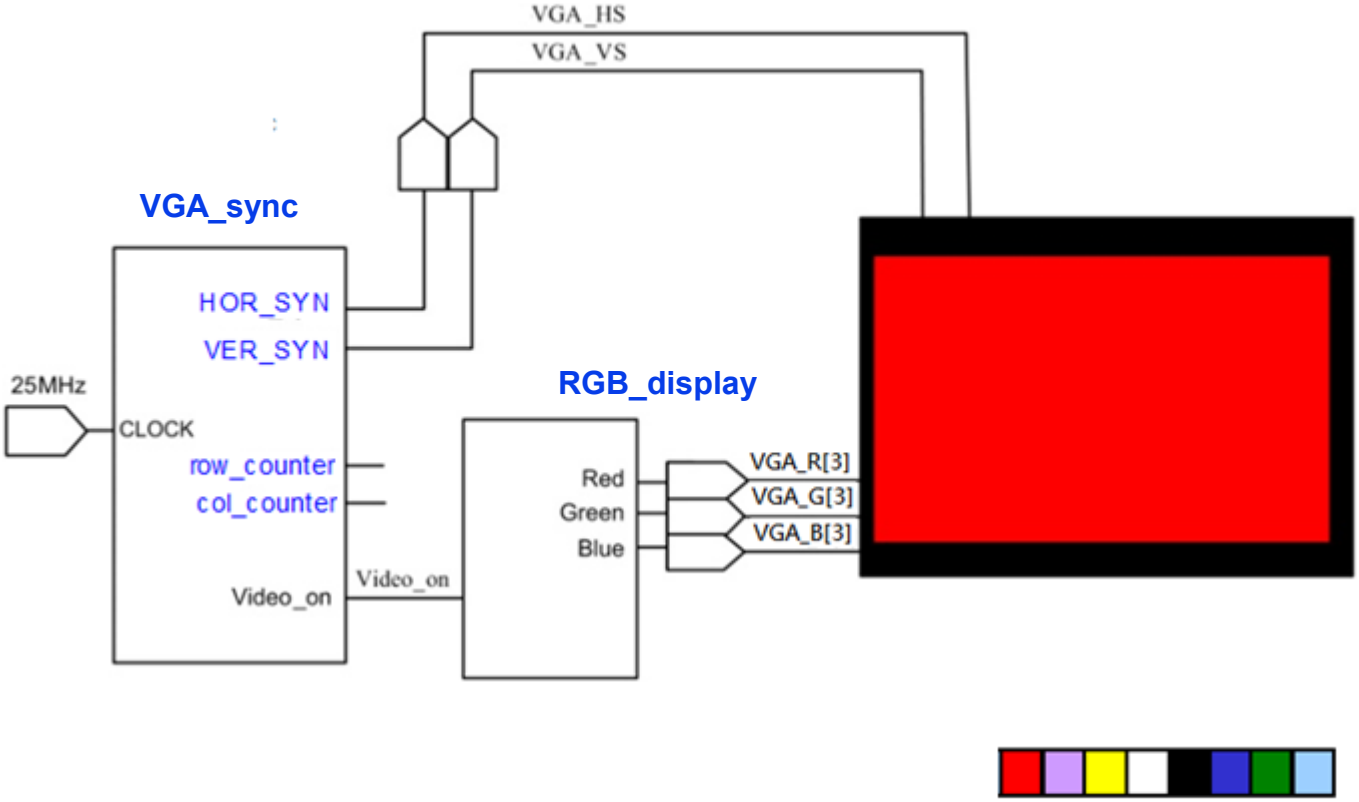

| Parameters | O | P | Q | R | S |
|---|---|---|---|---|---|
| Time | 16.68ms | 64us | 1.02ms | 15.25ms | 0.35ms |
| Lines | 525 | 2 | 30 | 480 | 14 |

# VGA Video on the DE0 Board

- There is a 4-bit DAC resistor network used to produce the analog data signals (red, green, and blue).

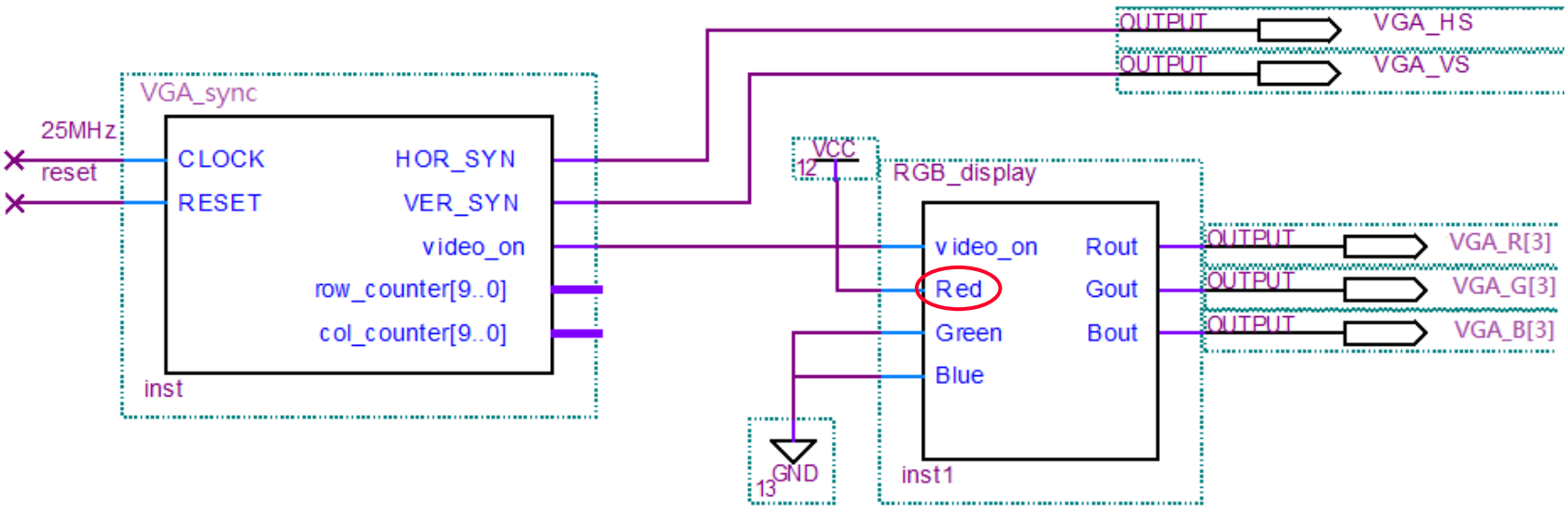


# 範例: 控制R,G,B輸入以在VGA螢幕上顯示八種不同的顏色

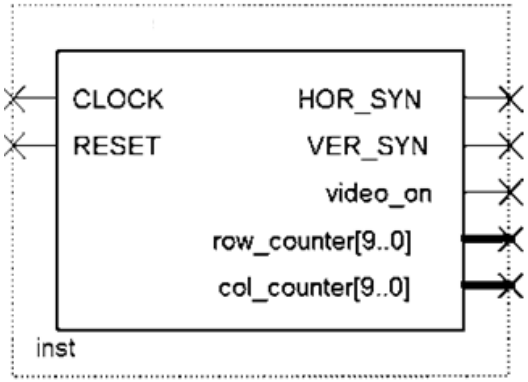


# 範例: 控制R,G,B輸入以在VGA螢幕上顯示八種不同的顏色

## <Block Diagram>



# VHDL: VGA_sync




```
Library ieee;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- Module Generates Video Sync Signals for Video Montor Interface
-- RGB and Sync outputs tie directly to monitor conector pins
ENTITY VGA_sync IS
    PORT(
        CLOCK,RESET: IN std_logic;
        HOR_SYN,VER_SYN,video_on: OUT std_logic;
        row_counter:out INTEGER RANGE 0 TO 524;
        col_counter:out INTEGER RANGE 0 TO 799  );
END VGA_drive ;

ARCHITECTURE arch OF VGA_drive IS
SIGNAL h_count: INTEGER RANGE 0 TO 799;
SIGNAL line_count: INTEGER RANGE 0 TO 524;
BEGIN
```

# VHDL: VGA_sync

```
--Generate Horizontal and Vertical Timing Signals for Video Signal
--
--  Horiz_sync  ------------------------------------__________--------
--  h_count     0                  639              660       755      799
--
    PROCESS(CLOCK,RESET)
    BEGIN
     IF RESET = '0' THEN  h_count <=0;
     ELSIF CLOCK'EVENT AND CLOCK='1' THEN
          IF h_count = 799 then h_count<=0;
          ELSE h_count <= h_count + 1;
          END IF;
     END IF;
    END PROCESS;

--Generate Horizontal Sync Signal using h_count
  PROCESS (h_count)
    BEGIN
        IF h_count >=660 and h_count<=755 THEN HOR_SYN <= '0';
        ELSE  HOR_SYN <= '1';
        END IF;
  END PROCESS;
```

# VHDL: VGA_sync

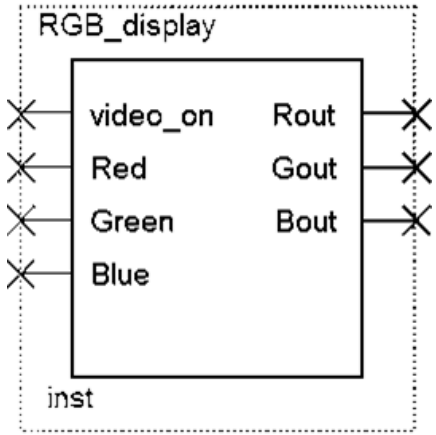
```
--  Vert_sync   ------------------------------------_______------------
--  line_count          0                479             493   494          524
--
    PROCESS(CLOCK,RESET)
     BEGIN
      IF RESET = '0' THEN line_count <=0;
      ELSIF CLOCK'EVENT AND CLOCK='1' THEN
          IF line_count = 799 then
             IF line_count = 524 THEN line_count <=0;
             ELSE line_count <= line_count+1;
             END IF;
          END IF;
     END IF;
     END PROCESS;


--Generate Vertical Sync Signal using line_count
    PROCESS (line_count)
     BEGIN
         IF (line_count >= 493 AND line_count <=494) THEN VER_SYN <= '0';
         ELSE VER_SYN <= '1';
         END IF;
     END PROCESS;
```

# VHDL: VGA_sync

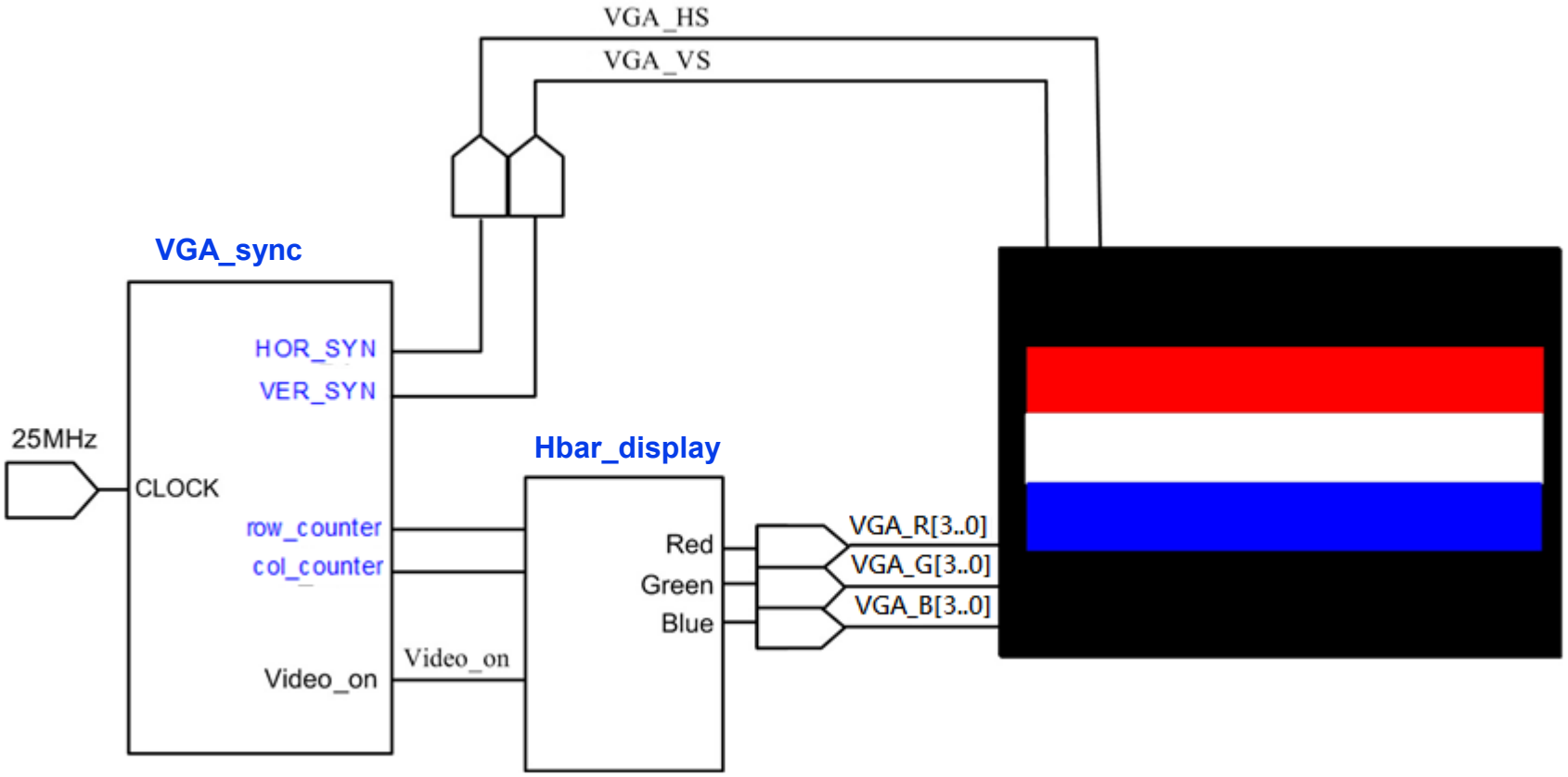
```
-- Generate Video on Screen Signals for Pixel Data
-- Video on = 1 indicates pixel are being displayed
  process (h_count, line_count)
    begin
        IF line_count >=480 and line_count<=524 THEN video_on<='0';
        ELSE
           IF h_count >=640 and h_count<=799 THEN video_on<='0';
           ELSE video_on<='1';
           END IF;
        END IF;
end process;

row_counter<=line_count;
col_counter<=h_count;

END arch;
```

# VHDL: RGB_display

```
Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


ENTITY RGB_display IS
    PORT(  video_on:IN std_logic;
        Red,Green,Blue: IN std_logic;
         Rout, Gout, Bout: out std_logic);
END RGB_display;

ARCHITECTURE arch OF RGB_display IS
begin

process(video_on)
begin
If video_on='1' then     --video time範圍內，RGB色彩依外界輸入設定而顯示
    Rout<=Red;
    Gout<=Green;
    Bout<=Blue;
else
    Rout<='0';Gout<='0';Bout<='0';     --video time以外，螢幕顯示全黑
end if;

end process;

END arch;
```

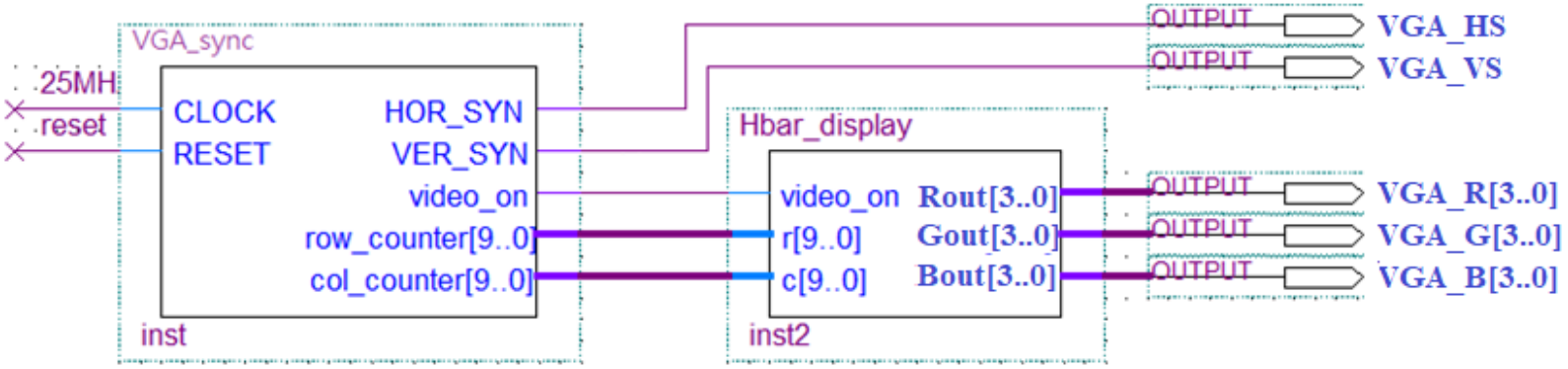


# 在VGA螢幕上顯示紅白藍橫條



# 在VGA螢幕上顯示紅白藍橫條

## <Block Diagram>



# VHDL: Hbar_display

```
Library IEEE;
use IEEE.STD_Logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY Hbar_display IS
    PORT( video_on:IN std_logic;
          r,c: IN std_logic_vector(9 downto 0);
          Rout, Gout, Bout: out std_logic_vector(3 downto 0));
END Hbar_display;

ARCHITECTURE arch OF Hbar_display IS
begin
process(video_on,r,c)
begin
If video_on='1' then          --設定行與列的範圍以顯示特定色彩
    IF (r>50 AND r<=150)and (c>0 and c<640) THEN
       Rout<="1111"; Gout<="0000"; Bout<="0000";
    else
       Rout<="0000"; Gout<="0000"; Bout<="0000";
    end if;

    IF (r>150 AND r<=250) and (c>0 and c<640) THEN
       Rout<="1111"; Gout<="1111"; Bout<="1111";
    else
          null;
    End if;

    IF (r>250 AND r<350) and (c>0 and c<640)  THEN
       Rout<="0000"; Gout<="0000"; Bout<="1111";
    else
         null;
    End if;
else                              --video time範圍以外全不顯示
    Rout<="0000"; Gout<="0000"; Bout<="0000";
end if;

end process;

END arch;
```

# VGA Timings

| Format | Pixel Clock (MHz) | Horizontal (in Pixels) | | | | Vertical (in Lines) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Active Video | Front Porch | Sync Pulse | Back Porch | Active Video | Front Porch | Sync Pulse | Back Porch |
| 640x480, 60Hz | 25.175 | 640 | 16 | 96 | 48 | 480 | 11 | 2 | 31 |
| 640x480, 72Hz | 31.500 | 640 | 24 | 40 | 128 | 480 | 9 | 3 | 28 |
| 640x480, 75Hz | 31.500 | 640 | 16 | 96 | 48 | 480 | 11 | 2 | 32 |
| 640x480, 85Hz | 36.000 | 640 | 32 | 48 | 112 | 480 | 1 | 3 | 25 |
| 800x600, 56Hz | 38.100 | 800 | 32 | 128 | 128 | 600 | 1 | 4 | 14 |
| 800x600, 60Hz | 40.000 | 800 | 40 | 128 | 88 | 600 | 1 | 4 | 23 |
| 800x600, 72Hz | 50.000 | 800 | 56 | 120 | 64 | 600 | 37 | 6 | 23 |
| 800x600, 75Hz | 49.500 | 800 | 16 | 80 | 160 | 600 | 1 | 2 | 21 |
| 800x600, 85Hz | 56.250 | 800 | 32 | 64 | 152 | 600 | 1 | 3 | 27 |
| 1024x768, 60Hz | 65.000 | 1024 | 24 | 136 | 160 | 768 | 3 | 6 | 29 |
| 1024x768, 70Hz | 75.000 | 1024 | 24 | 136 | 144 | 768 | 3 | 6 | 29 |
| 1024x768, 75Hz | 78.750 | 1024 | 16 | 96 | 176 | 768 | 1 | 3 | 28 |
| 1024x768, 85Hz | 94.500 | 1024 | 48 | 96 | 208 | 768 | 1 | 3 | 36 |
| 1280x1024, 60Hz | 108.00 | 1280 | 48 | 112 | 248 | 768 | 1 | 3 | 38 |
| HDMI 720P 1280x720p 60Hz | 75.25 | 1280 | 72 | 80 | 216 | 720 | 3 | 5 | 30 |
| HDMI 1080P 1920x1040 60Hz | 148.5 | 1920 | 88 | 44 | 148 | 1080 | 4 | 5 | 36 |

Source: Rick Ballantyne, Xilinx Inc.