Serial Protocool

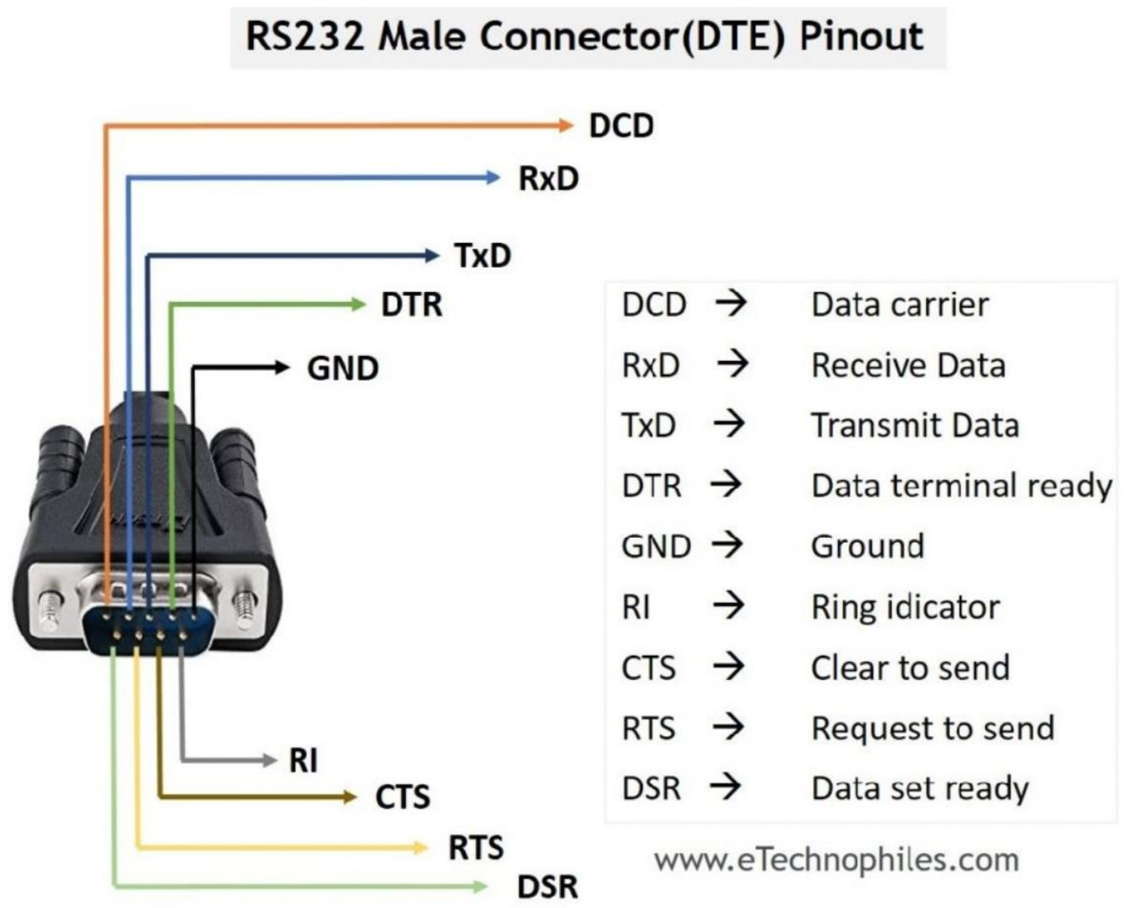
Agenda

- Uart_Loopback
- SPI_Flash
- IIC_EEPROM

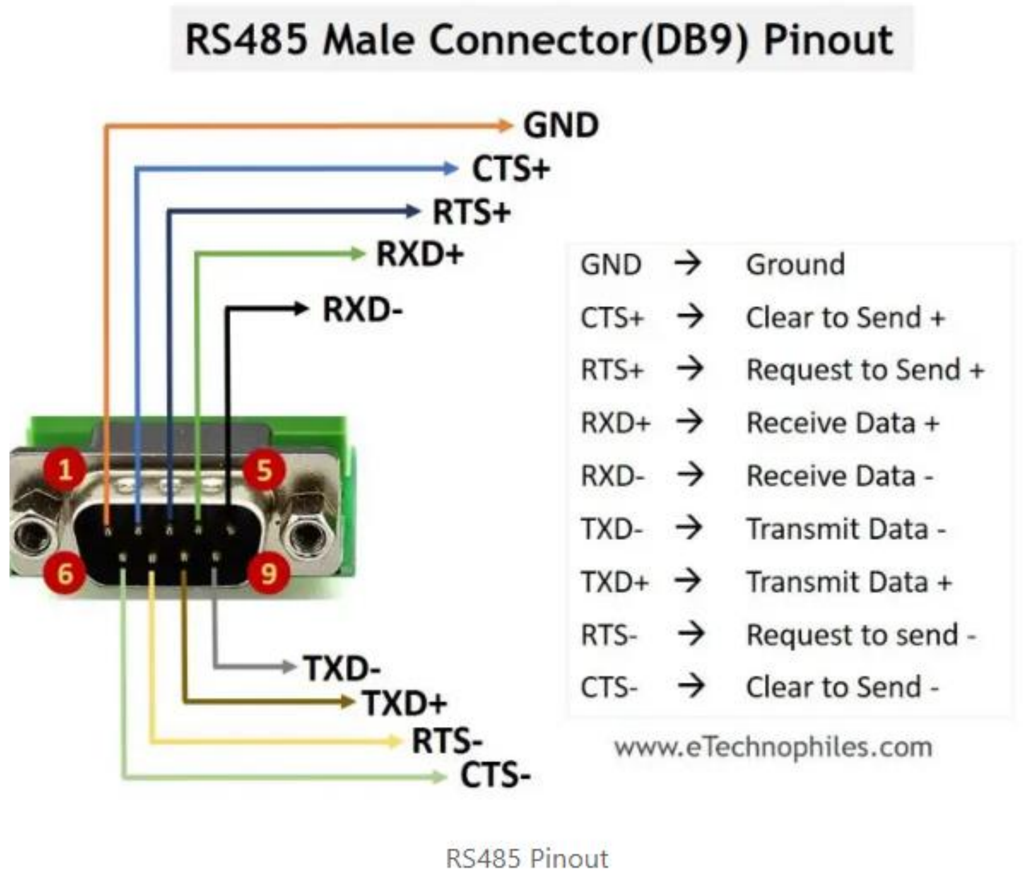
UART

UART overview

全稱Universal Asynchronous Receiver/Transmitter，屬於異步Serial通信接口的統稱。
後續會做一個簡單的UART_LOOP實現Lab



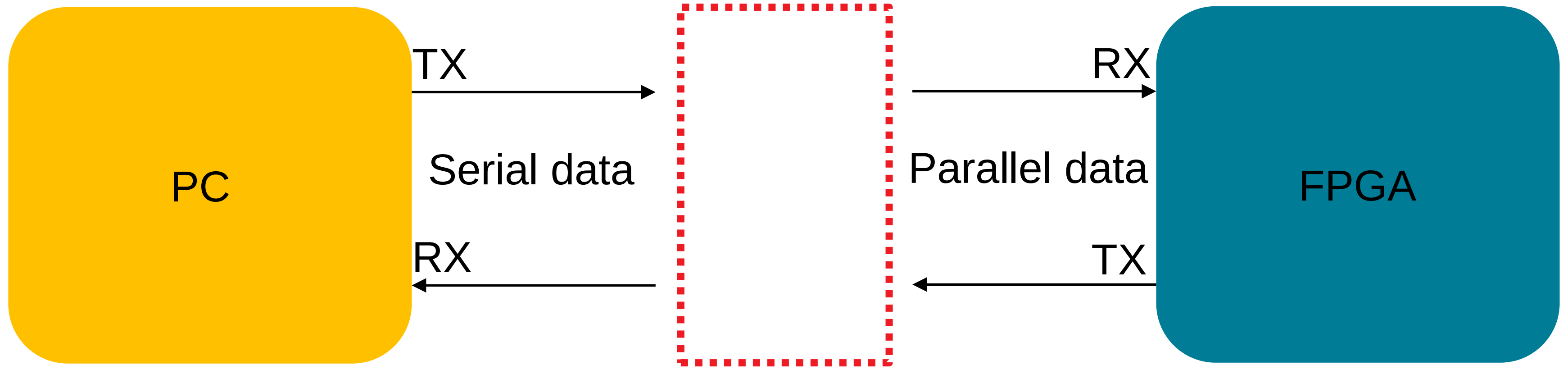
DCD	→	Data carrier
RxD	→	Receive Data
TxD	→	Transmit Data
DTR	→	Data terminal ready
GND	→	Ground
RI	→	Ring idicator
CTS	→	Clear to send
RTS	→	Request to send
DSR	→	Data set ready

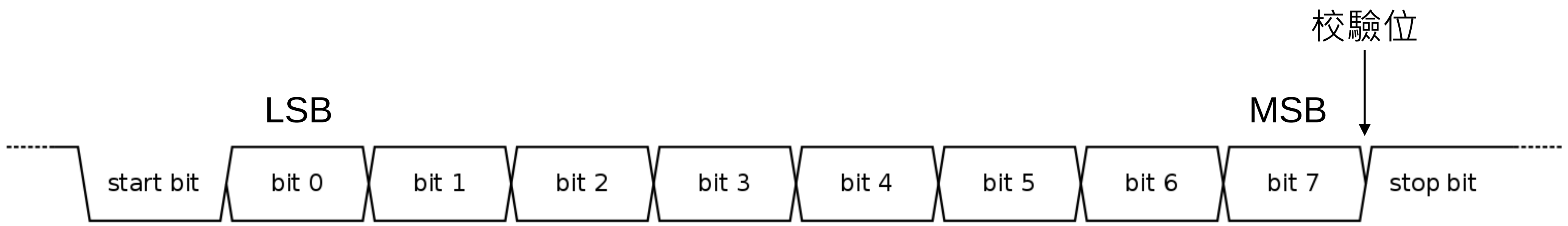


GND	→	Ground
CTS+	→	Clear to Send +
RTS+	→	Request to Send +
RXD+	→	Receive Data +
RXD-	→	Receive Data -
TXD-	→	Transmit Data -
TXD+	→	Transmit Data +
RTS-	→	Request to send -
CTS-	→	Clear to Send -

RS232

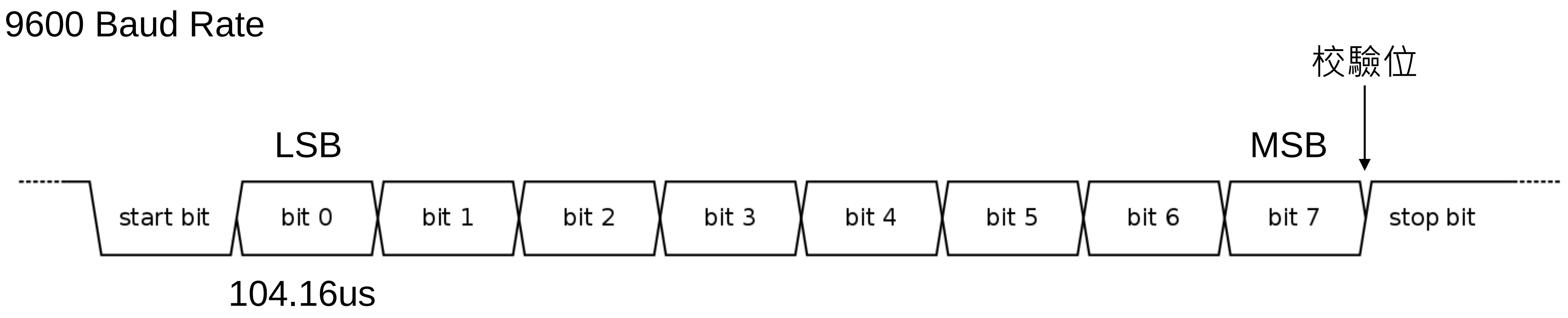
RS232是全雙工(可同時讀寫)，並且使用單端信號



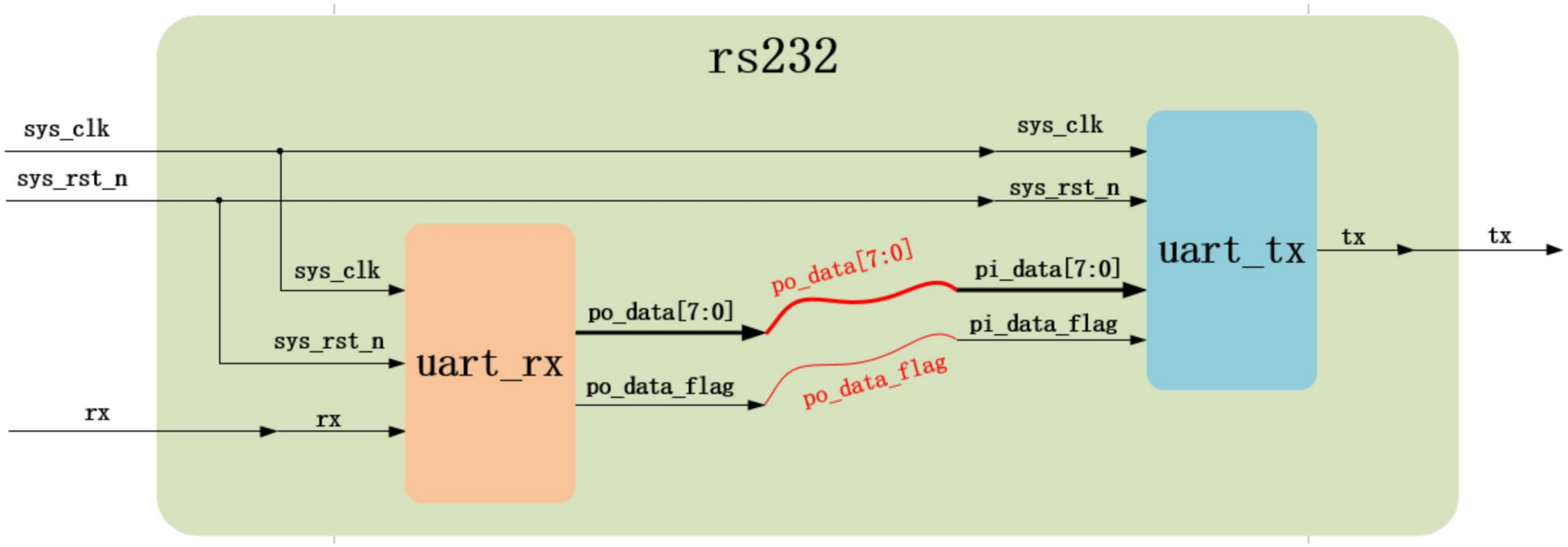


Baud Rate

以9600 Baud Rate為例，每秒傳送9600個bit 即 一個bit時間是104.16us。
且因為UART屬於異步通訊所以需要考量系統時鐘頻率。
BAUD_CNT_MAX = CLK_PERIOD / BAUD_BPS
舉例來說，如果時鐘頻率為50MHZ 配上9600 Baud Rate則一個bit位會花費約5208個時鐘週期



RS232 Hardware architecture



uart_rx

將Baud Rate 及時鐘週期拉出來方便更改

```
module uart_rx#(
    parameter UART_BPS = 'd9600, // Baud Rate
    parameter CLK_FREQ = 'd50_000_000 // Clock period
    )(
    input wire sys_clk , //  50MHz
    input wire sys_rst_n ,
    input wire rx , // Receive Data
    output reg [7:0] po_data , // Parallel Data
    output reg po_flag
    );
```

uart_rx

BAUD_CNT_MAX 代表一個bit花費的時鐘週期

```
//localparam define
localparam BAUD_CNT_MAX = CLK_FREQ/UART_BPS ;
//reg define
reg rx_reg1 ;
reg rx_reg2 ;
reg rx_reg3 ;
reg start_flag ;
reg work_en ;
reg [12:0] baud_cnt ;
reg bit_flag ;
reg [3:0] bit_cnt ;
reg [7:0] rx_data ;
reg rx_flag ;
```

uart_rx

reg1讓兩邊的clock同步，後面兩個rx_reg是為了抑制抖動消除亞穩態

```
//  Synchronize Clock Period
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        rx_reg1 <= 1'b1;
    else
        rx_reg1 <= rx;

// Eliminate metastability
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        rx_reg2 <= 1'b1;
    else
        rx_reg2 <= rx_reg1;
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        rx_reg3 <= 1'b1;
    else
        rx_reg3 <= rx_reg2;
```

uart_rx

判斷真正的開始位

```
//  Edge Detect
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        start_flag <= 1'b0;
    else if((~rx_reg2) && (rx_reg3) && (work_en == 1'b0))
        start_flag <= 1'b1;
    else
        start_flag <= 1'b0;
```

uart_rx

判斷一次傳輸的bit數為8位

```
//  Detect Recive 8 bit Rx_data
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        work_en <= 1'b0;
    else if(start_flag == 1'b1)
        work_en <= 1'b1;
    else if((bit_cnt == 4'd8) && (bit_flag == 1'b1))
        work_en <= 1'b0;
    else
        work_en <= work_en;
// Count receive 8 bit Rx_data
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        bit_cnt <= 4'b0;
    else if((bit_cnt == 4'd8) && (bit_flag == 1'b1))
        bit_cnt <= 4'b0;
    else if(bit_flag ==1'b1)
        bit_cnt <= bit_cnt + 1'b1;
```

uart_rx

Baud rate的計數器，並抓到一個bit的時間區間正中間來傳輸

```
// Baud count  1bit 5208 clock period
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        baud_cnt <= 13'b0;
    else if((baud_cnt == BAUD_CNT_MAX - 1) || (work_en == 1'b0))
        baud_cnt <= 13'b0;
    else if(work_en == 1'b1)
        baud_cnt <= baud_cnt + 1'b1;
// Search middle time interval
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        bit_flag <= 1'b0;
    else if(baud_cnt == BAUD_CNT_MAX/2 - 1)
        bit_flag <= 1'b1;
    else
        bit_flag <= 1'b0;
```

uart_rx

將Serial Data 轉為 Parallel Data 需要透過一個移位寄存器來拼接

```
 // Shift Rx_data
 always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        rx_data <= 8'b0;
    else if((bit_cnt >= 4'd1)&&(bit_cnt <= 4'd8)&&(bit_flag ==
1'b1))
        rx_data <= {rx_reg3, rx_data[7:1]};

 //  Shift Rx_data flag
 always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        rx_flag <= 1'b0;
    else if((bit_cnt == 4'd8) && (bit_flag == 1'b1))
        rx_flag <= 1'b1;
    else
        rx_flag <= 1'b0;
```

uart_rx

最後輸出data ， 最後為了同步時鐘週期所以打了一拍

```
  // Output Data
 always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        po_data <= 8'b0;
    else if(rx_flag == 1'b1)
        po_data <= rx_data;

 // Output Data flag
 always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        po_flag <= 1'b0;
    else
        po_flag <= rx_flag;

endmodule
```

tb_uart_rx

Tb需要給生成一組隨機的輸入資料

```
 `timescale 1ns/1ns
module tb_uart_rx();

//reg define
reg sys_clk;
reg sys_rst_n;
reg rx;
 //wire define
wire [7:0] po_data;
wire po_flag;
 // initial
 initial begin
    sys_clk = 1'b1;
    sys_rst_n <= 1'b0;
    rx <= 1'b1;
    #20;
    sys_rst_n <= 1'b1;
 end
```

tb_uart_rx

透過task來反覆傳送，0位起始 9位停止

```
//  serial 8 bit  input data
 initial begin
    #200
    rx_bit(8'd0);
    rx_bit(8'd1);
    rx_bit(8'd2);
    rx_bit(8'd3);
    rx_bit(8'd4);
    rx_bit(8'd5);
    rx_bit(8'd6);
    rx_bit(8'd7);
 end

 always #10 sys_clk = ~sys_clk;
```

```
task rx_bit(
    input [7:0] data
);
integer i;
for(i=0; i<10; i=i+1) begin
    case(i)
         0: rx <= 1'b0;
         1: rx <= data[0];
         2: rx <= data[1];
         3: rx <= data[2];
         4: rx <= data[3];
         5: rx <= data[4];
         6: rx <= data[5];
         7: rx <= data[6];
         8: rx <= data[7];
         9: rx <= 1'b1;
    endcase
         #(5208*20);
    end
endtask
```

tb_uart_rx

實例化

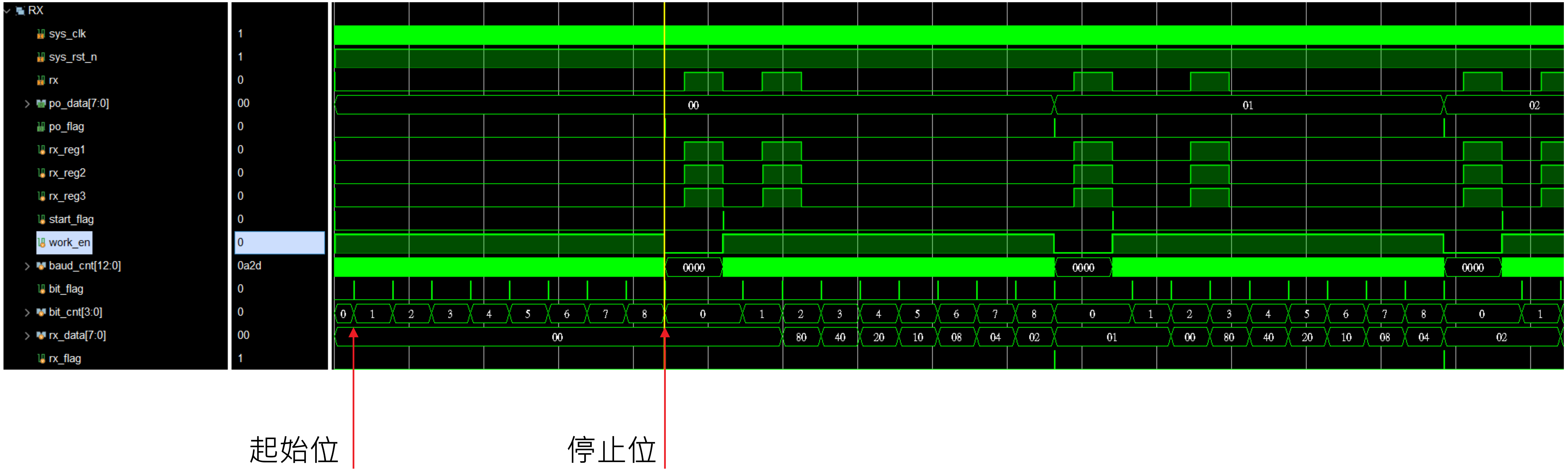
```
//------------------------uart_rx_inst------------------------
uart_rx
#(
.UART_BPS(9600),
.CLK_FREQ(50_000_000))
uart_rx_inst(
.sys_clk (sys_clk ), //input sys_clk
.sys_rst_n (sys_rst_n ), //input sys_rst_n
.rx (rx ), //input rx

.po_data (po_data ), //output [7:0] po_data
.po_flag (po_flag ) //output po_flag
);

endmodule
```

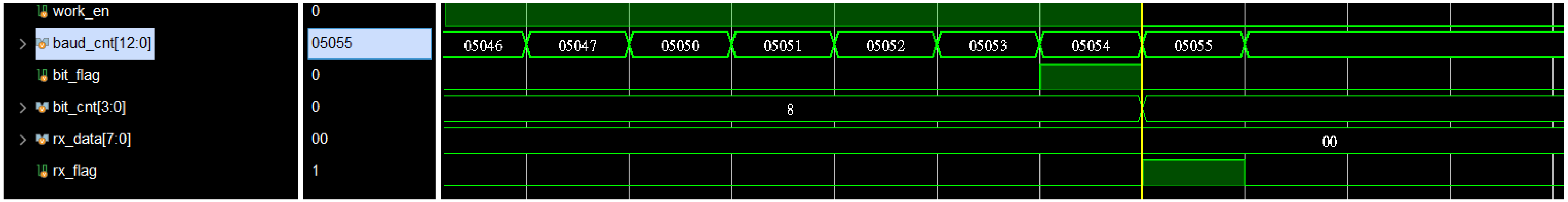
Simulation

依據各家模擬器不同，rx_reg1.2.3不一定觀察的到clock週期的延遲

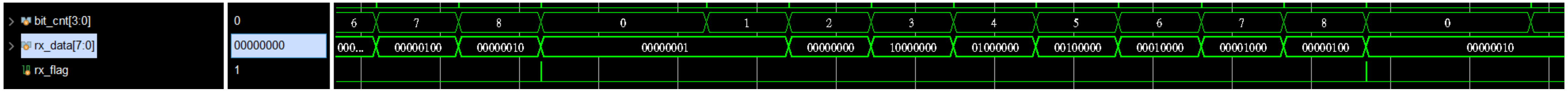


Simulation

Baud Rate計數



rx_dara 移位拼接



AMD
together we advance_

uart_tx

將Baud Rate 及時鐘週期拉出來方便更改

```
module uart_tx#(
    parameter UART_BPS = 'd9600, // Baud Rate
    parameter CLK_FREQ = 'd50_000_000 // clock period
)(
    input wire sys_clk ,
    input wire sys_rst_n ,
    input wire [7:0] pi_data ,
    input wire pi_flag ,
    output reg tx
 );
 //localparam define
 localparam BAUD_CNT_MAX = CLK_FREQ/UART_BPS ;
 //reg define
 reg [12:0] baud_cnt;
 reg bit_flag;
 reg [3:0] bit_cnt ;
 reg work_en ;
```

uart_tx

將原本的parallel data轉為10位(起始位 8bit數據位 停止位) 傳輸

```
// enable signal
always@(posedge sys_clk or negedge sys_rst_n)
   if(sys_rst_n == 1'b0)
       work_en <= 1'b0;
   else if(pi_flag == 1'b1)
       work_en <= 1'b1;
   else if((bit_flag == 1'b1) && (bit_cnt == 4'd9))
       work_en <= 1'b0;
// bit cnt  8 + 2bit
always@(posedge sys_clk or negedge sys_rst_n)
   if(sys_rst_n == 1'b0)
       bit_cnt <= 4'b0;
   else if((bit_flag == 1'b1) && (bit_cnt == 4'd9))
       bit_cnt <= 4'b0;
   else if((bit_flag == 1'b1) && (work_en == 1'b1))
       bit_cnt <= bit_cnt + 1'b1;
```

uart_tx

Baud rate計算

```
 // baud_cnt
always@(posedge sys_clk or negedge sys_rst_n)
   if(sys_rst_n == 1'b0)
       baud_cnt <= 13'b0;
   else if((baud_cnt == BAUD_CNT_MAX - 1) || (work_en == 1'b0))
       baud_cnt <= 13'b0;
   else if(work_en == 1'b1)
       baud_cnt <= baud_cnt + 1'b1;

// bit flag
always@(posedge sys_clk or negedge sys_rst_n)
   if(sys_rst_n == 1'b0)
       bit_flag <= 1'b0;
   else if(baud_cnt == 13'd1)
       bit_flag <= 1'b1;
   else
       bit_flag <= 1'b0;
```

uart_tx

利用case 將資料轉成serial傳輸

```
// parallel to serial
always@(posedge sys_clk or negedge sys_rst_n)
   if(sys_rst_n == 1'b0)
       tx <= 1'b1;
   else if(bit_flag == 1'b1)
       case(bit_cnt)
           0 : tx <= 1'b0;
           1 : tx <= pi_data[0];
           2 : tx <= pi_data[1];
           3 : tx <= pi_data[2];
           4 : tx <= pi_data[3];
           5 : tx <= pi_data[4];
           6 : tx <= pi_data[5];
           7 : tx <= pi_data[6];
           8 : tx <= pi_data[7];
           9 : tx <= 1'b1;
           default : tx <= 1'b1;
       endcase
endmodule
```

AMD
together we advance_

tb_uart_tx

一樣需要模擬輸入資料

```
module tb_uart_tx();
//reg define
reg sys_clk;
reg sys_rst_n;
reg [7:0] pi_data;
reg pi_flag;
 //wire define
 wire tx;
 // initial
 initial begin
    sys_clk = 1'b1;
    sys_rst_n <= 1'b0;
    #20;
    sys_rst_n <= 1'b1;
 end
```

tb_uart_tx

每個sys_clk週期就傳一bit資料

```
// output [7:0]data
 initial begin
    pi_data <= 8'b0;
    pi_flag <= 1'b0;
    #200
    pi_data <= 8'd0;
    pi_flag <= 1'b1;
    #20
    pi_flag <= 1'b0;
    #(5208*20*10);
    pi_data <= 8'd1;
    pi_flag <= 1'b1;
    #20
    pi_flag <= 1'b0;
     #(5208*20*10);
    pi_data <= 8'd2;
    pi_flag <= 1'b1;
```

```
    #20
    pi_flag <= 1'b0;
    #(5208*20*10);
    pi_data <= 8'd3;
    pi_flag <= 1'b1;
    #20
    pi_flag <= 1'b0;
    #(5208*20*10);
    pi_data <= 8'd4;
    pi_flag <= 1'b1;
    #20
    pi_flag <= 1'b0;
    #(5208*20*10);
    pi_data <= 8'd5;
    pi_flag <= 1'b1;
    #20
    pi_flag <= 1'b0;
```

```
    #(5208*20*10);
    pi_data <= 8'd6;
    pi_flag <= 1'b1;
    #20
    pi_flag <= 1'b0;
    #(5208*20*10);
    pi_data <= 8'd7;
    pi_flag <= 1'b1;
    #20
    pi_flag <= 1'b0;
    end
 always #10 sys_clk =
~sys_clk;
```

AMD
together we advance_

rs232

將之前的rx.tx實例化

```
 module rs232(
input wire sys_clk ,
input wire sys_rst_n ,
input wire rx ,
output wire tx
);

 //parameter define
 parameter UART_BPS = 14'd9600;
 parameter CLK_FREQ = 26'd50_000_000;

 //wire define
 wire [7:0] po_data;
 wire po_flag;
```

rs232

```
 //------------------------uart_rx_inst------------------------
uart_rx
 #(
 .UART_BPS (UART_BPS),
 .CLK_FREQ (CLK_FREQ)
 )
 uart_rx_inst
 (
 .sys_clk (sys_clk ), //input sys_clk
 .sys_rst_n (sys_rst_n ), //input sys_rst_n
 .rx (rx ), //input rx

 .po_data (po_data ), //output [7:0] po_data
 .po_flag (po_flag ) //output po_flag
 );
```

rs232

```
//-------------------------uart_tx_inst------------------------
 uart_tx
 #(
 .UART_BPS (UART_BPS),
 .CLK_FREQ (CLK_FREQ)
 )
 uart_tx_inst
 (
 .sys_clk (sys_clk ), //input sys_clk
 .sys_rst_n (sys_rst_n ), //input sys_rst_n
 .pi_data (po_data ), //input [7:0] pi_data
 .pi_flag (po_flag ), //input pi_flag

 .tx (tx ) //output tx
 );
 endmodule
```

tb_rs232

Tb需要給生成一組隨機的輸入資料

```
 `timescale 1ns/1ns
module tb_rs232();
reg sys_clk;
reg sys_rst_n;
reg rx;
wire tx;

 initial begin
    sys_clk = 1'b1;
    sys_rst_n <= 1'b0;
    rx <= 1'b1;
    #20;
    sys_rst_n <= 1'b1;
 end
 initial begin
 #200
 rx_byte();
 end
```

AMD together we advance_

tb_rs232

透過task來反覆傳送，0位起始 9位停止

```
always #10 sys_clk = ~sys_clk;

 task rx_byte();
 integer j;
 for(j=0; j<8; j=j+1)
 rx_bit(j);
 endtask
 task rx_bit(
 input [7:0] data
 );
 integer i;
```

```
  for(i=0; i<10; i=i+1) begin
  case(i)
  0: rx <= 1'b0;
  1: rx <= data[0];
  2: rx <= data[1];
  3: rx <= data[2];
  4: rx <= data[3];
  5: rx <= data[4];
  6: rx <= data[5];
  7: rx <= data[6];
  8: rx <= data[7];
  9: rx <= 1'b1;
  endcase
  #(5208*20);
  end
  endtask
```

[Public]

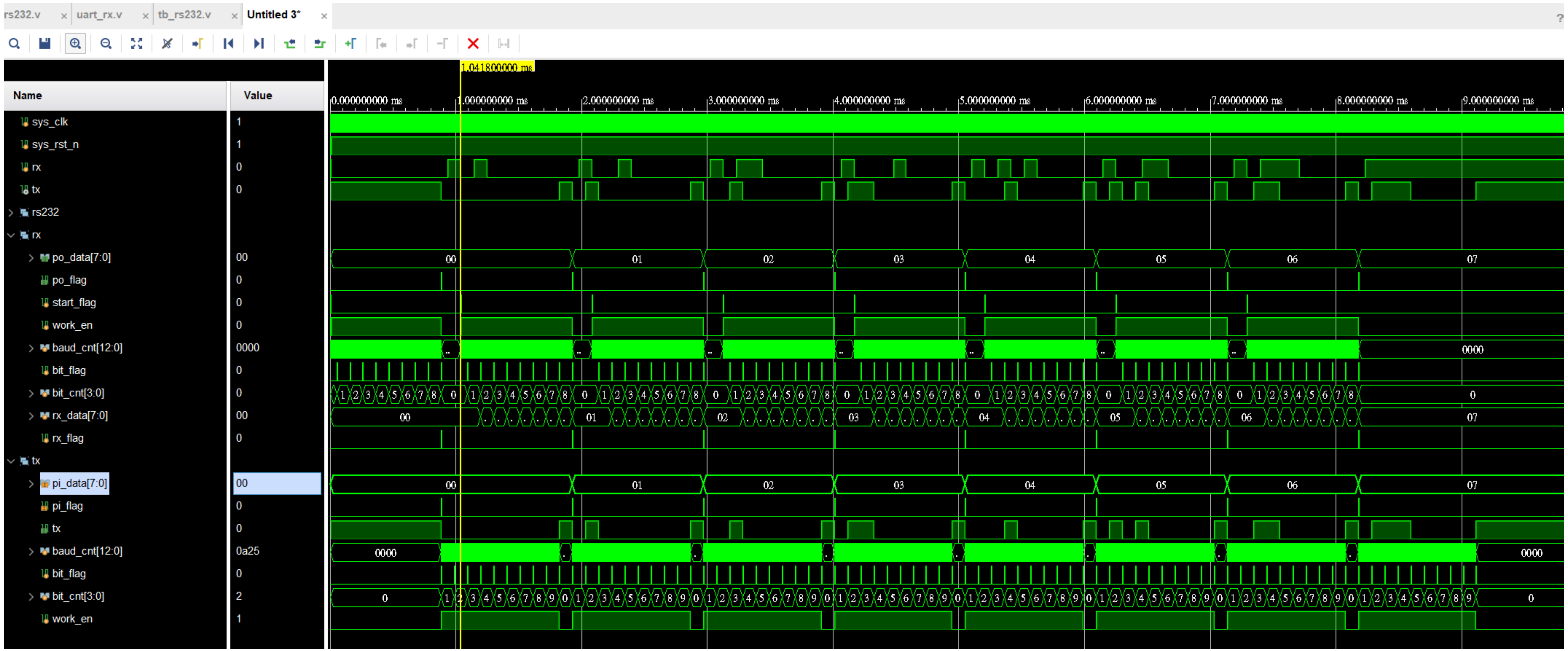
tb_rs232

實例化

```
//------------------------rs232_inst------------------------
rs232 rs232_inst(
.sys_clk (sys_clk ), //input sys_clk
.sys_rst_n (sys_rst_n ), //input sys_rst_n
.rx (rx ), //input rx
.tx (tx ) //output tx
);

endmodule
```

rs232



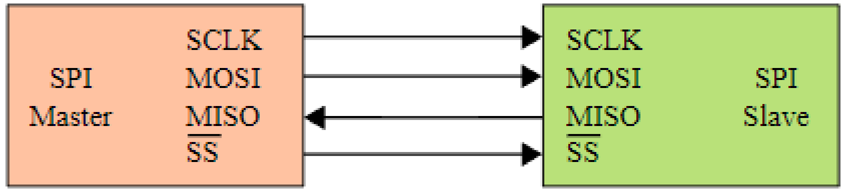
Name	Value
sys_clk	1
sys_rst_n	1
rx	0
tx	0
rs232	
rx	
po_data[7:0]	00
po_flag	0
start_flag	0
work_en	0
baud_cnt[12:0]	0000
bit_flag	0
bit_cnt[3:0]	0
rx_data[7:0]	00
rx_flag	0
tx	
pi_data[7:0]	00
pi_flag	0
tx	0
baud_cnt[12:0]	0a25
bit_flag	0
bit_cnt[3:0]	2
work_en	1

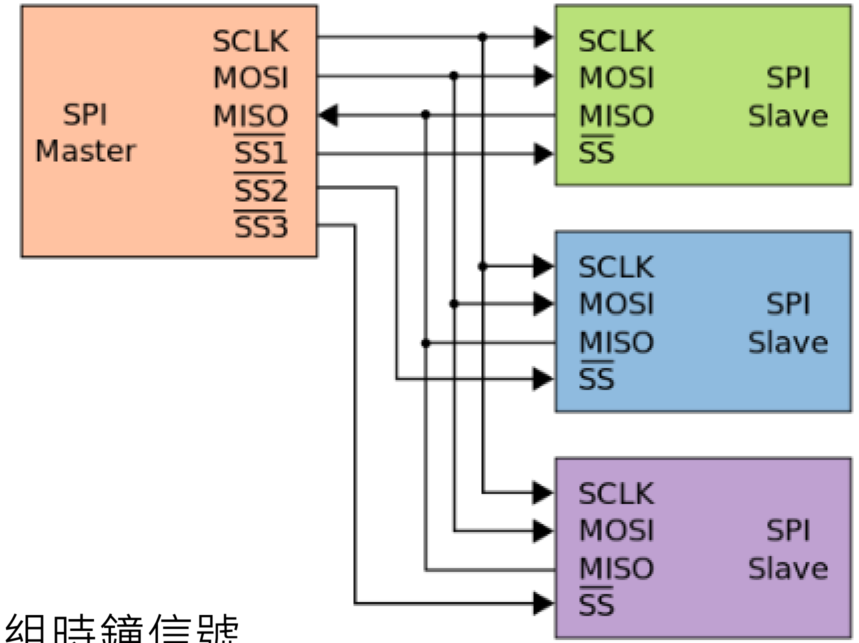
AMD
together we advance_

SPI

SPI overview

SPI (Serial Periphel Interface) 為一種高速全雙工的同步通訊協議，但沒有應答機制確認數據是否接受。
SPI有主從之分，下圖例分別顯示單對單及單對多的情形

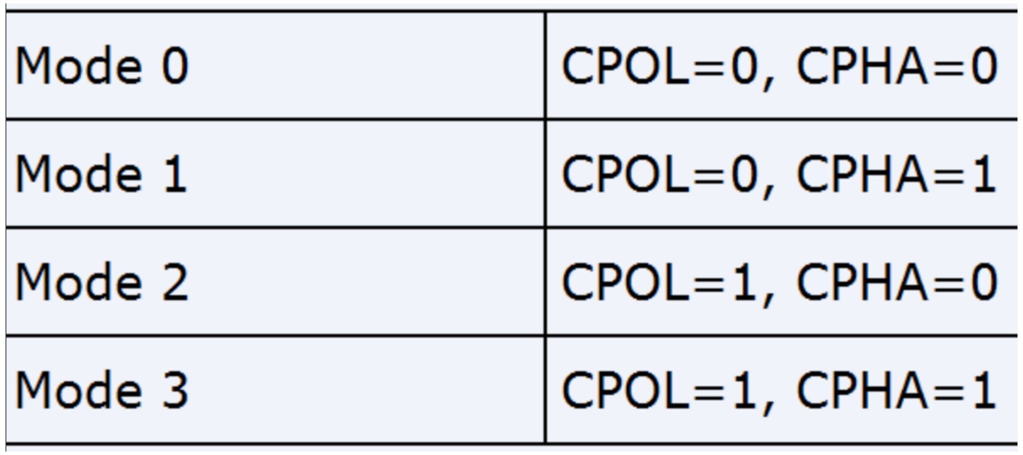




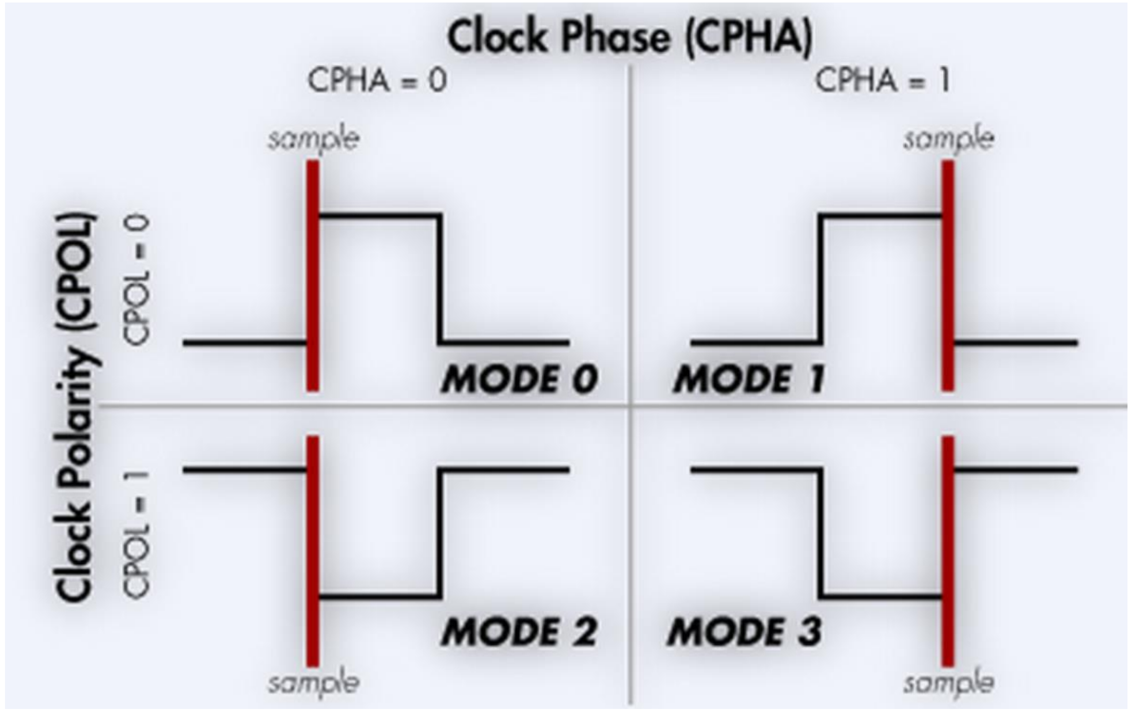
SCLK : 因為是同步通訊，所以使用的是同一組時鐘信號
MOSI : 主機輸出 / 從機輸入
MISO : 主機輸入 / 從機輸出
SS : Chip Select 有幾個從機就需要幾個訊號線

SPI 協議

實際看到協議前，會先將clock區分成極性(CPOL)及相位(CPHA)



Mode 0	CPOL=0, CPHA=0
Mode 1	CPOL=0, CPHA=1
Mode 2	CPOL=1, CPHA=0
Mode 3	CPOL=1, CPHA=1

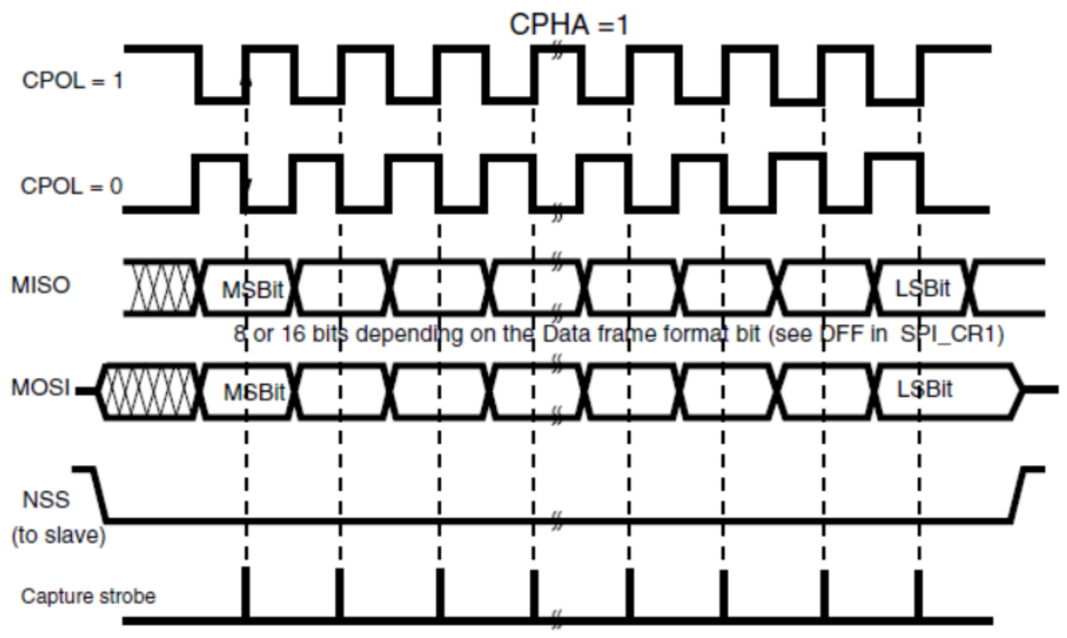


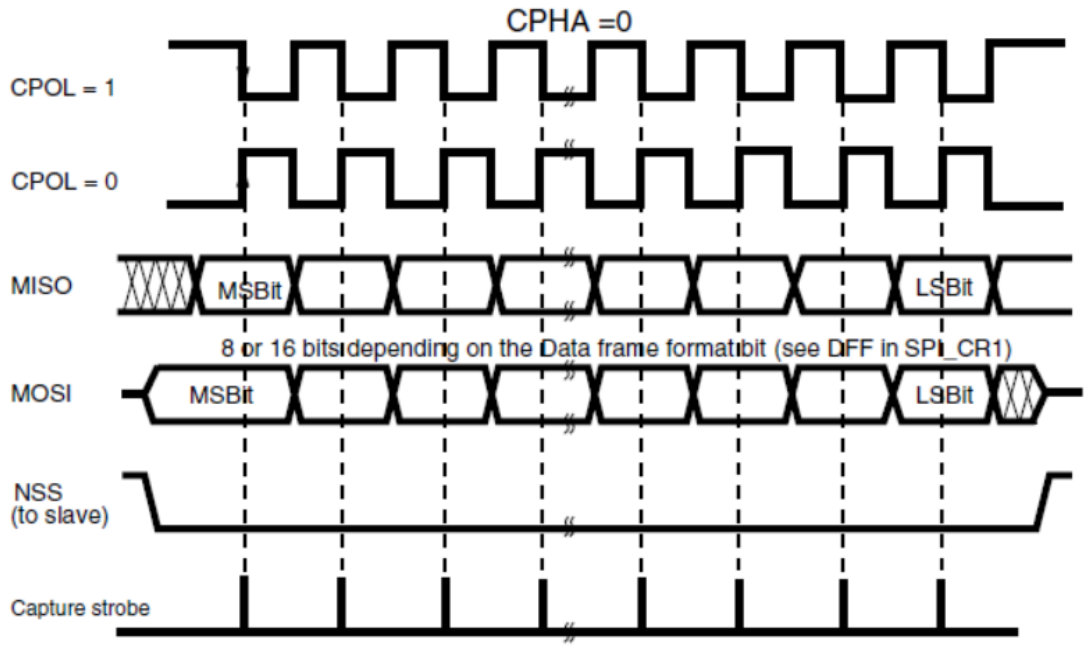
當CPOL為0時
CPHA為0 代表奇數位採樣
CPHA為1 代表偶數位採樣

當CPOL為1時
CPHA為0 代表偶數位採樣
CPHA為1 代表奇數位採樣

SPI 協議

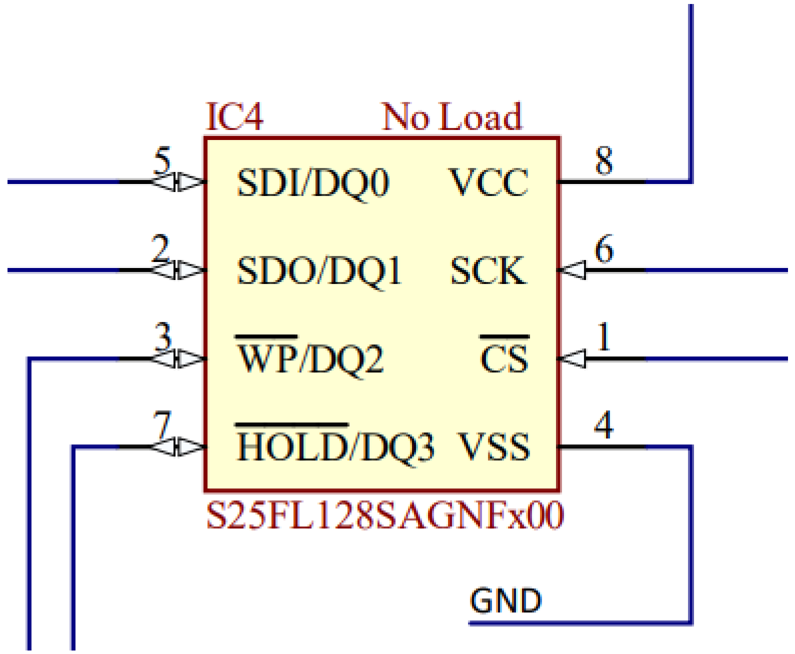
可以看到四種模式的時序圖，當SS拉低認到從機時就會依據模式判定的時機抓取資料



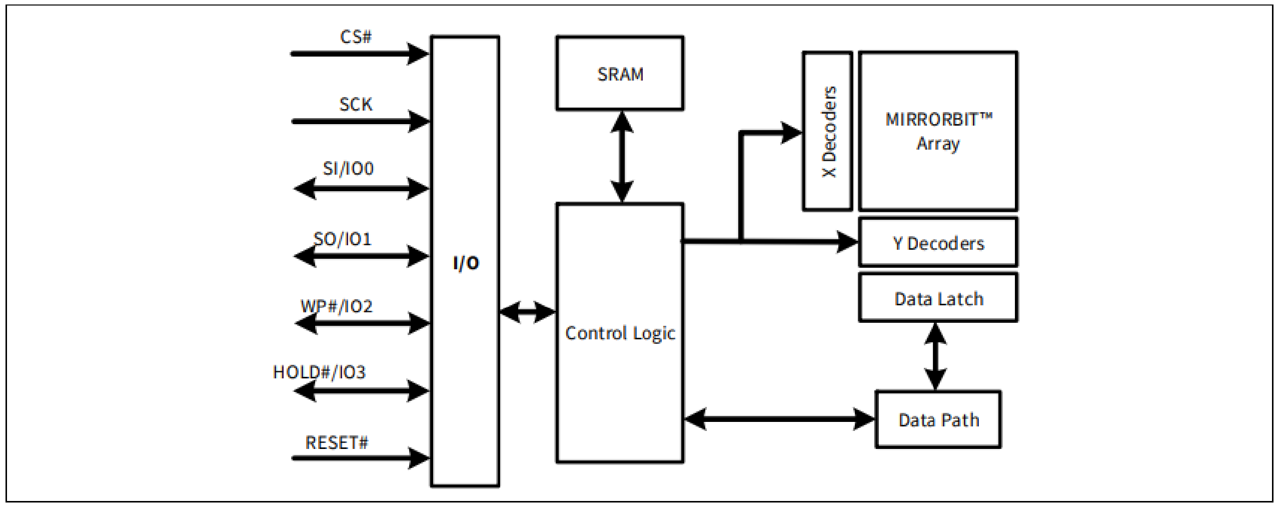


Flash_Erase

FPGA通常是透過搭配的FLASH來固化程序，所以這裡會準備幾個SPI FLASH的Lab來學習如何讀寫控制FLASH
本次Lab使用Arty S7-50 搭配的S25FL128S，相關資料參閱其Datasheet。
首先我們先將學習自己編寫程式將SPI FLASH內的資料消除



Logic block diagram



Flash_Erase

The Bulk Erase (BE) command sets all bits to '1' (all bytes are FFh) inside the entire flash memory array. Before the BE command can be accepted by the device, a Write Enable (WREN) command must be issued and decoded by the device, which sets the Write Enable Latch (WEL) in the Status Register to enable any write operations.

CS# must be driven into the logic HIGH state after the eighth bit of the instruction byte has been latched in on SI. This will initiate the erase cycle, which involves the pre-programming and erase of the entire flash memory array. If CS# is not driven HIGH after the last bit of instruction, the BE operation will not be executed.

As soon as CS# is driven into the logic HIGH state, the erase cycle will be initiated. With the erase cycle in progress, the user can read the value of the Write-In Progress (WIP) bit to determine when the operation has been completed. The WIP bit will indicate a '1' when the erase cycle is in progress and a '0' when the erase cycle has been completed.

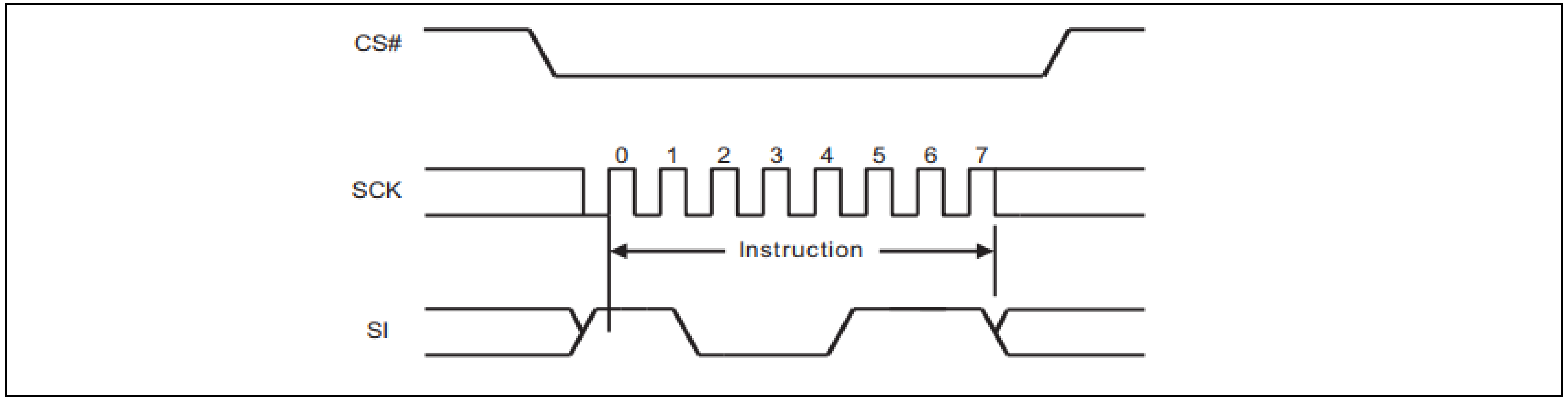
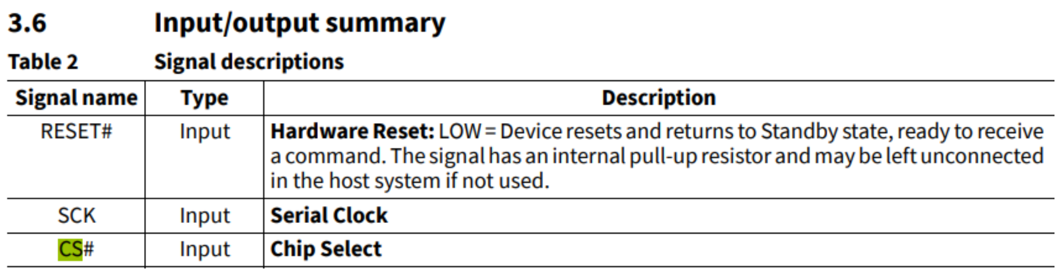


Figure 110 **Bulk Erase command sequence**

1. 拉高WREN
2. 進入WEL 狀態
3. 拉低CS信號並持續
4. 將每個bit寫為1
5. 拉高CS信號
6. 等待erase cycle的時間

Flash_Erase



3.6 Input/output summary

Table 2 Signal descriptions

Signal name	Type	Description
RESET#	Input	**Hardware Reset:** LOW = Device resets and returns to Standby state, ready to receive a command. The signal has an internal pull-up resistor and may be left unconnected in the host system if not used.
SCK	Input	**Serial Clock**
CS#	Input	**Chip Select**

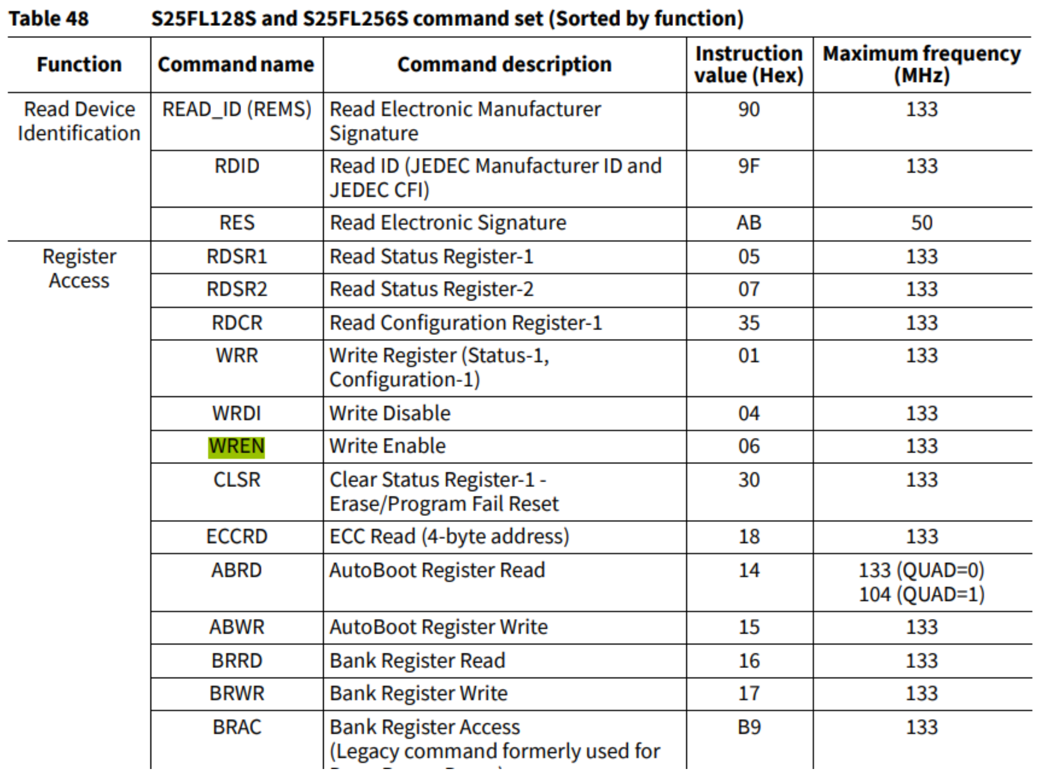


Table 48 S25FL128S and S25FL256S command set (Sorted by function)

Function	Command name	Command description	Instruction value (Hex)	Maximum frequency (MHz)
Read Device Identification	READ_ID (REMS)	Read Electronic Manufacturer Signature	90	133
	RDID	Read ID (JEDEC Manufacturer ID and JEDEC CFI)	9F	133
	RES	Read Electronic Signature	AB	50
Register Access	RDSR1	Read Status Register-1	05	133
	RDSR2	Read Status Register-2	07	133
	RDCR	Read Configuration Register-1	35	133
	WRR	Write Register (Status-1, Configuration-1)	01	133
	WRDI	Write Disable	04	133
	WREN	Write Enable	06	133
	CLSR	Clear Status Register-1 - Erase/Program Fail Reset	30	133
	ECCRD	ECC Read (4-byte address)	18	133
	ABRD	AutoBoot Register Read	14	133 (QUAD=0) 104 (QUAD=1)
	ABWR	AutoBoot Register Write	15	133
	BRRD	Bank Register Read	16	133
	BRWR	Bank Register Write	17	133
	BRAC	Bank Register Access (Legacy command formerly used for	B9	133

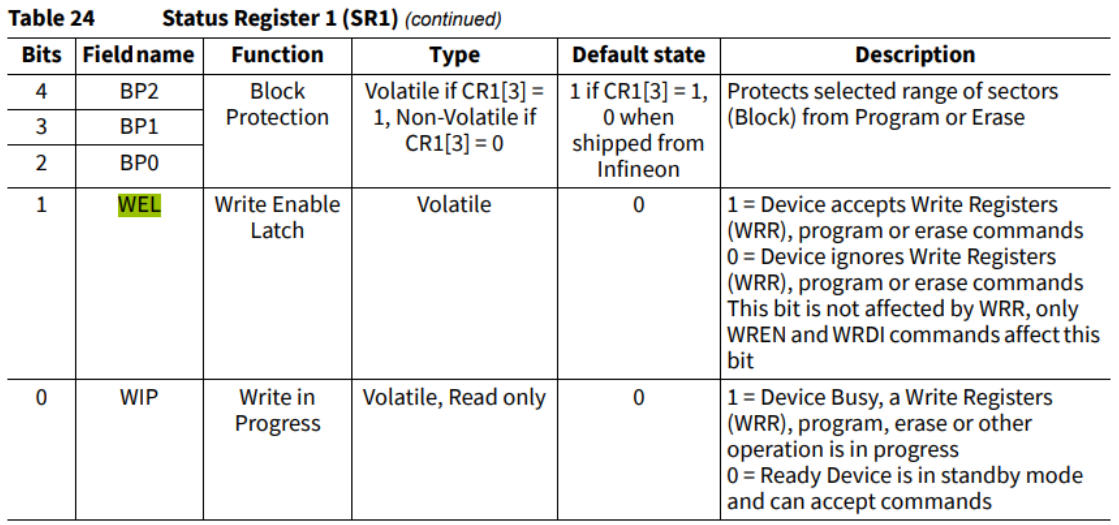


Table 24 Status Register 1 (SR1) *(continued)*

Bits	Field name	Function	Type	Default state	Description
4	BP2	Block Protection	Volatile if CR1[3] = 1, Non-Volatile if CR1[3] = 0	1 if CR1[3] = 1, 0 when shipped from Infineon	Protects selected range of sectors (Block) from Program or Erase
3	BP1				
2	BP0				
1	WEL	Write Enable Latch	Volatile	0	1 = Device accepts Write Registers (WRR), program or erase commands 0 = Device ignores Write Registers (WRR), program or erase commands This bit is not affected by WRR, only WREN and WRDI commands affect this bit
0	WIP	Write in Progress	Volatile, Read only	0	1 = Device Busy, a Write Registers (WRR), program, erase or other operation is in progress 0 = Ready Device is in standby mode and can accept commands

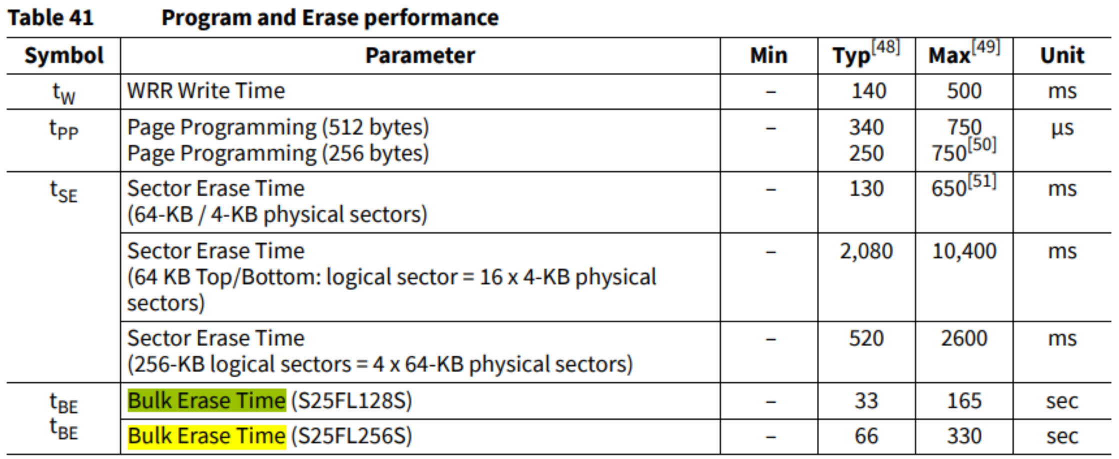


Table 41 Program and Erase performance

Symbol	Parameter	Min	Typ[48]	Max[49]	Unit
t_W	WRR Write Time	–	140	500	ms
t_{PP}	Page Programming (512 bytes) Page Programming (256 bytes)	–	340 250	750 750[50]	µs
t_{SE}	Sector Erase Time (64-KB / 4-KB physical sectors)	–	130	650[51]	ms
	Sector Erase Time (64 KB Top/Bottom: logical sector = 16 x 4-KB physical sectors)	–	2,080	10,400	ms
	Sector Erase Time (256-KB logical sectors = 4 x 64-KB physical sectors)	–	520	2600	ms
t_{BE}	Bulk Erase Time (S25FL128S)	–	33	165	sec
t_{BE}	Bulk Erase Time (S25FL256S)	–	66	330	sec

Timing

6.4.1 Clock timing

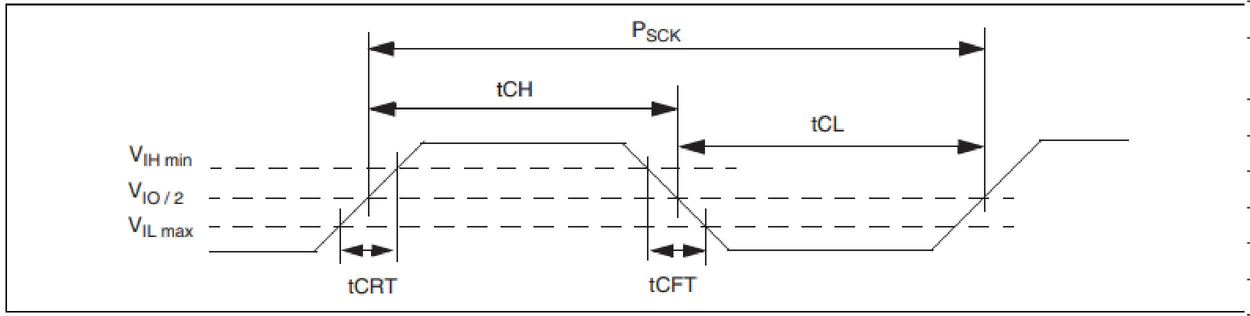


Figure 34 Clock timing

6.4.2 Input / output timing

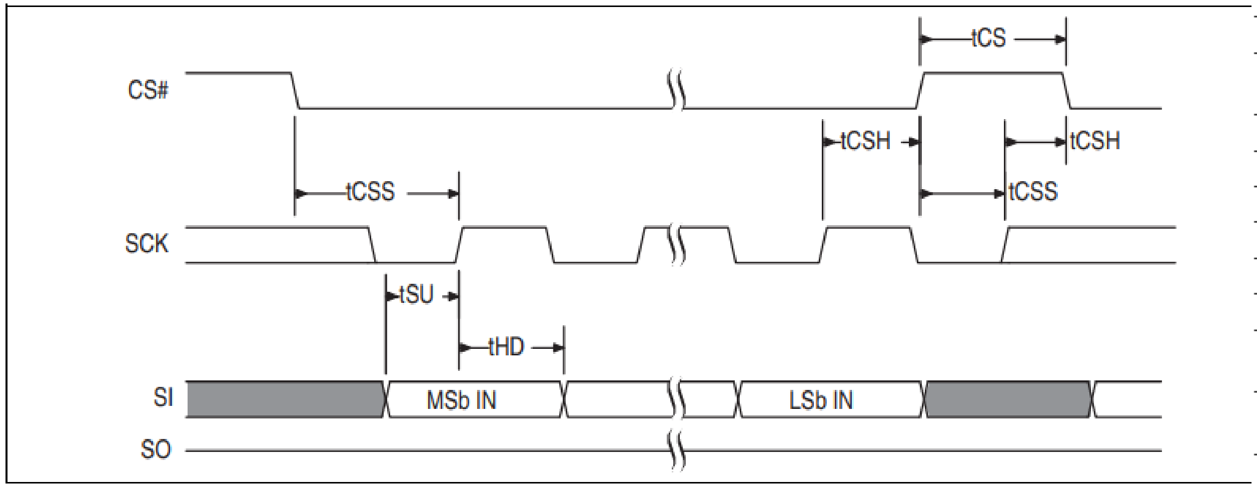
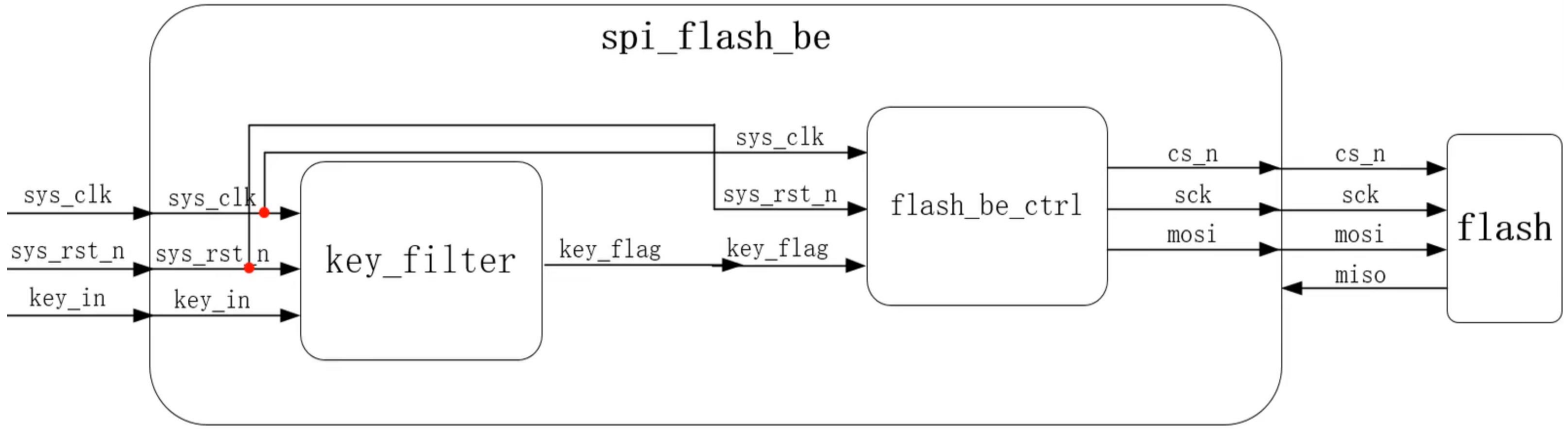


Figure 35 SPI single bit input timing

Table 14 AC characteristics (Single die package, V_{IO} 1.65 V to 2.7 V, V_{CC} 2.7 V to 3.6 V)

Symbol	Parameter	Min	Typ	Max	Unit
$F_{SCK, R}$	SCK clock frequency for READ, 4READ instructions	DC	–	50	MHz
$F_{SCK, C}$	SCK clock frequency for all others[34]	DC	–	66	MHz
P_{SCK}	SCK clock period	$1/ F_{SCK}$	–	∞	
t_{WH}, t_{CH}	Clock high time[35]	45% P_{SCK}	–	–	ns
t_{WL}, t_{CL}	Clock low time[35]	45% P_{SCK}	–	–	ns
t_{CRT}, t_{CLCH}	Clock rise time (slew rate)	0.1	–	–	V/ns
t_{CFT}, t_{CHCL}	Clock fall time (slew rate)	0.1	–	–	V/ns
t_{CS}	CS# high time (Read instructions) CS# high time (Program/erase)	10 50	–	–	ns
t_{CSS}	CS# active setup time (relative to SCK)	10	–	–	ns
t_{CSH}	CS# active hold time (relative to SCK)	3	–	–	ns
t_{SU}	Data in setup time	5	–	3000[36]	ns
t_{HD}	Data in hold time	4	–	–	ns
t_V	Clock low to output valid	–	–	14.5[33] 12.0[34]	ns
t_{HO}	Output hold time	2	–		ns
t_{DIS}	Output disable time	0	–	14	ns
t_{WPS}	WP# setup time	20[32]	–	–	ns
t_{WPH}	WP# hold time	100[32]	–	–	ns
t_{HLCH}	HOLD# active setup time (relative to SCK)	5	–	–	ns
t_{CHHH}	HOLD# active hold time (relative to SCK)	5	–	–	ns
t_{HHCH}	HOLD# non active setup time (relative to SCK)	5	–	–	ns
t_{CHHL}	HOLD# non active hold time (relative to SCK)	5	–	–	ns
t_{HZ}	HOLD# enable to output invalid	–	–	14	ns
t_{LZ}	HOLD# disable to output valid	–	–	14	ns

Bulk Erase Lab



Spi_flash_be

依據Flash Datasheet指示，若要清除Flash，需要先進寫入狀態

```
module flash_be_ctrl
(
input wire sys_clk ,
input wire sys_rst_n ,
input wire key ,
output reg cs_n ,
output reg sck ,
output reg mosi
);
parameter IDLE = 4'b0001 , WR_EN = 4'b0010 , DELAY = 4'b0100 , BE = 4'b1000 ;
parameter WR_EN_INST = 8'b0000_0110, BE_INST = 8'b1100_0111;
//reg define
reg [2:0] cnt_byte; // byte counter
reg [3:0] state ; // state
reg [4:0] cnt_clk ; // sys_clk counter
reg [1:0] cnt_sck ; // serial clock counter
reg [2:0] cnt_bit ; // bit counter
```

Spi_flash_be

Sys_clk使用12.5MHz 依據Flash Datasheet給的時間計算，一個bit花費32個clk週期

```
// cnt_clk:
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        cnt_clk <= 5'd0;
    else if(state != IDLE)
        cnt_clk <= cnt_clk + 1'b1;
//cnt_byte: 1byte 32 clock period
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        cnt_byte <= 3'd0;
    else if((cnt_clk == 5'd31) && (cnt_byte == 3'd6))
        cnt_byte <= 3'd0;
    else if(cnt_clk == 31)
        cnt_byte <= cnt_byte + 1'b1;
```

Spi_flash_be

依據剛剛產生的flag 來產生SPI的serial clock

```
//cnt_sck: serial clock counter
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        cnt_sck <= 2'd0;
    else if((state == WR_EN) && (cnt_byte == 1'b1))
        cnt_sck <= cnt_sck + 1'b1;
    else if((state == BE) && (cnt_byte == 3'd5))
        cnt_sck <= cnt_sck + 1'b1;

//sck: output serial clock
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        sck <= 1'b0;
    else if(cnt_sck == 2'd0)
        sck <= 1'b0;
    else if(cnt_sck == 2'd2)
        sck <= 1'b1;
```

Spi_flash_be

控制cs信號來輸出mosi

```
//cs_n:
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        cs_n <= 1'b1;
    else if(key == 1'b1)
        cs_n <= 1'b0;
    else if((cnt_byte == 3'd2)&&(cnt_clk == 5'd31)&&(state == WR_EN))
        cs_n <= 1'b1;
    else if((cnt_byte == 3'd3)&&(cnt_clk == 5'd31)&&(state == DELAY))
        cs_n <= 1'b0;
    else if((cnt_byte == 3'd6)&&(cnt_clk == 5'd31)&&(state == BE))
        cs_n <= 1'b1;
//cnt_bit:output mosi
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        cnt_bit <= 3'd0;
    else if(cnt_sck == 2'd2)
        cnt_bit <= cnt_bit + 1'b1;
```

Spi_flash_be

狀態機 狀態轉移模塊

```
//state machine
always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        state <= IDLE;
    else
    case(state)
        IDLE: if(key == 1'b1)
                state <= WR_EN;
        WR_EN: if((cnt_byte == 3'd2) && (cnt_clk == 5'd31))
                state <= DELAY;
        DELAY: if((cnt_byte == 3'd3) && (cnt_clk == 5'd31))
                state <= BE;
        BE: if((cnt_byte == 3'd6) && (cnt_clk == 5'd31))
                state <= IDLE;
        default: state <= IDLE;
    endcase
```

Spi_flash_be

狀態機 輸出模塊

```
//mosi: output
 always@(posedge sys_clk or negedge sys_rst_n)
    if(sys_rst_n == 1'b0)
        mosi <= 1'b0;
    else if((state == WR_EN) && (cnt_byte == 3'd2))
        mosi <= 1'b0;
    else if((state == BE) && (cnt_byte == 3'd6))
        mosi <= 1'b0;
    else if((state == WR_EN)&&(cnt_byte == 3'd1)&&(cnt_sck == 5'd0))
        mosi <= WR_EN_INST[7 - cnt_bit];
    else if((state == BE) && (cnt_byte == 3'd5) && (cnt_sck == 5'd0))
        mosi <= BE_INST[7 - cnt_bit];

 endmodule
```

tb_flash_be_ctrl

```
`timescale 1ns/1ns
module tb_flash_be_ctrl();

//wire define
wire cs_n ;
wire sck ;
wire mosi ;

//reg define
 reg sys_clk ;
 reg sys_rst_n ;
 reg key ;
```

```
initial
 begin
    sys_clk = 1'b1;
    sys_rst_n <= 1'b0;
    key <= 1'b0;
    #100
    sys_rst_n <= 1'b1;
    #1000
    key <= 1'b1;
    #20
    key <= 1'b0;
end

 always #10 sys_clk <= ~sys_clk;

 defparam memory.mem_access.initfile = "initmemory.txt";
```

tb_flash_be_ctrl

```
//-- flash_be_ctrl_inst -------
flash_be_ctrl flash_be_ctrl_inst
(
.sys_clk (sys_clk ),
.sys_rst_n (sys_rst_n ),
.key (key ),
.sck (sck ),
.cs_n (cs_n ),
.mosi (mosi )
);
```

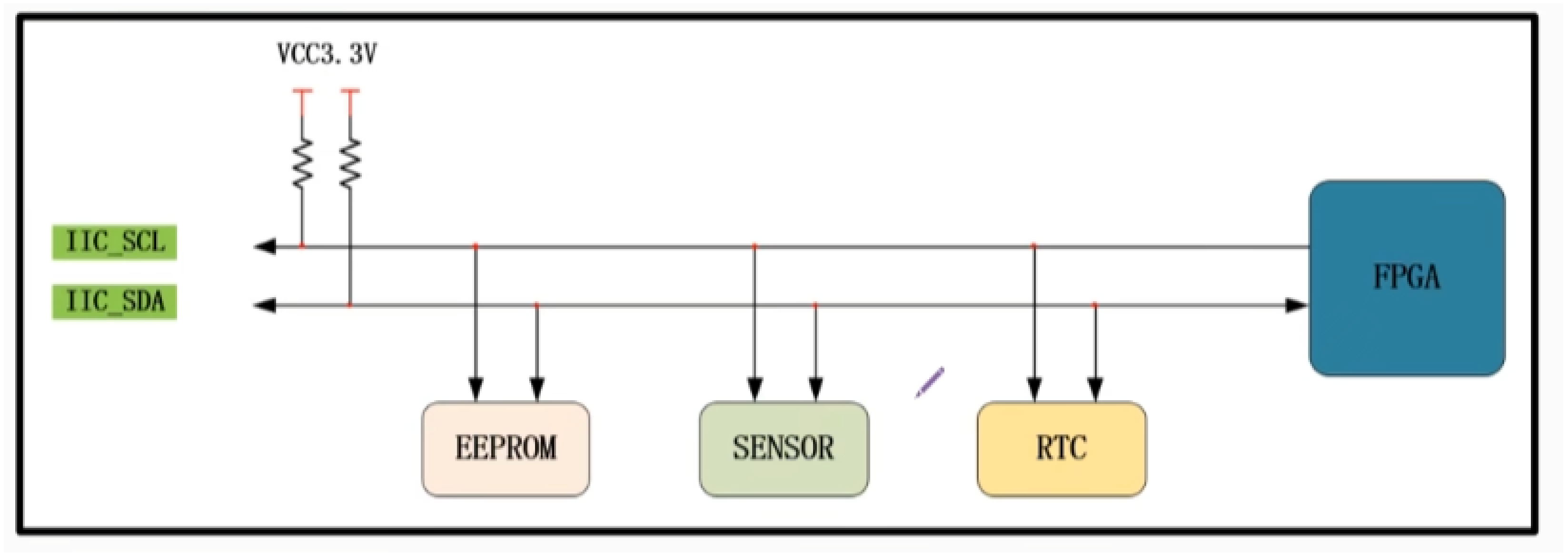
```
//------------- memory ------------
 m25p16 memory
 (
 .c (sck ),
 .data_in (mosi ),
 .s (cs_n ),
 .w (1'b1 ),
 .hold (1'b1 ),
 .data_out ( )
 );

 endmodule
```

IIC

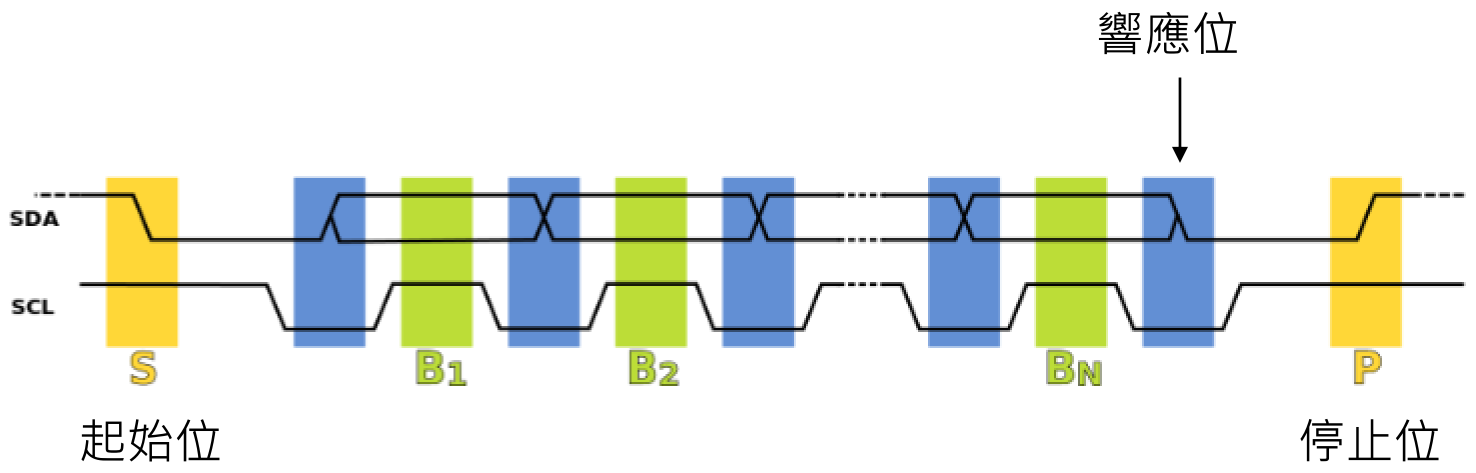
IIC overview

全稱Inter – Integrated Circuit，屬於同步雙向Serial通信接口的統稱。
而且總共只需要兩線即可傳輸 SCL負責同步 SDA負責傳輸數據
支援多主機對多從機，



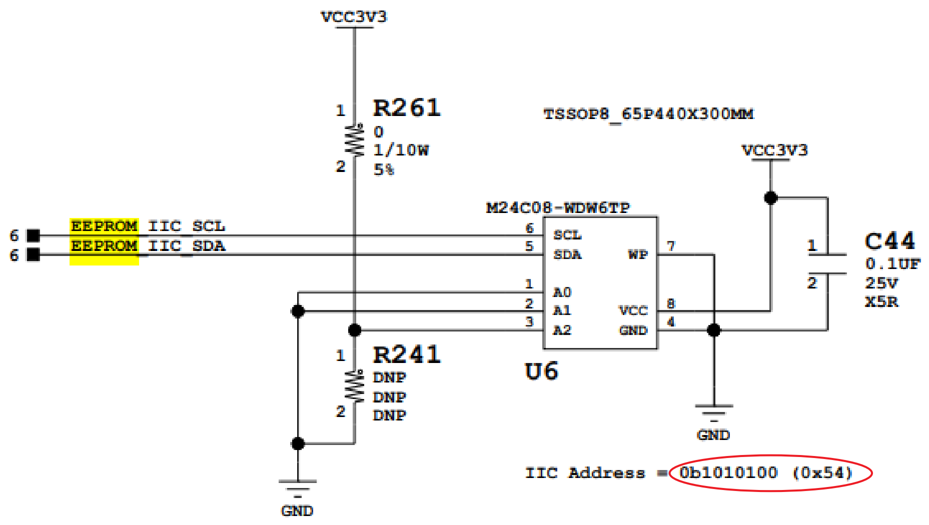
AMD
together we advance_

IIC 協議層



IIC Address

以AC701使用的IIC EEPROM為例，每種IIC的從機設備不同，以EEPROM來說只啟用了低3位
主機在傳輸時會先辨識從機的address，若兩相符合才拉高響應



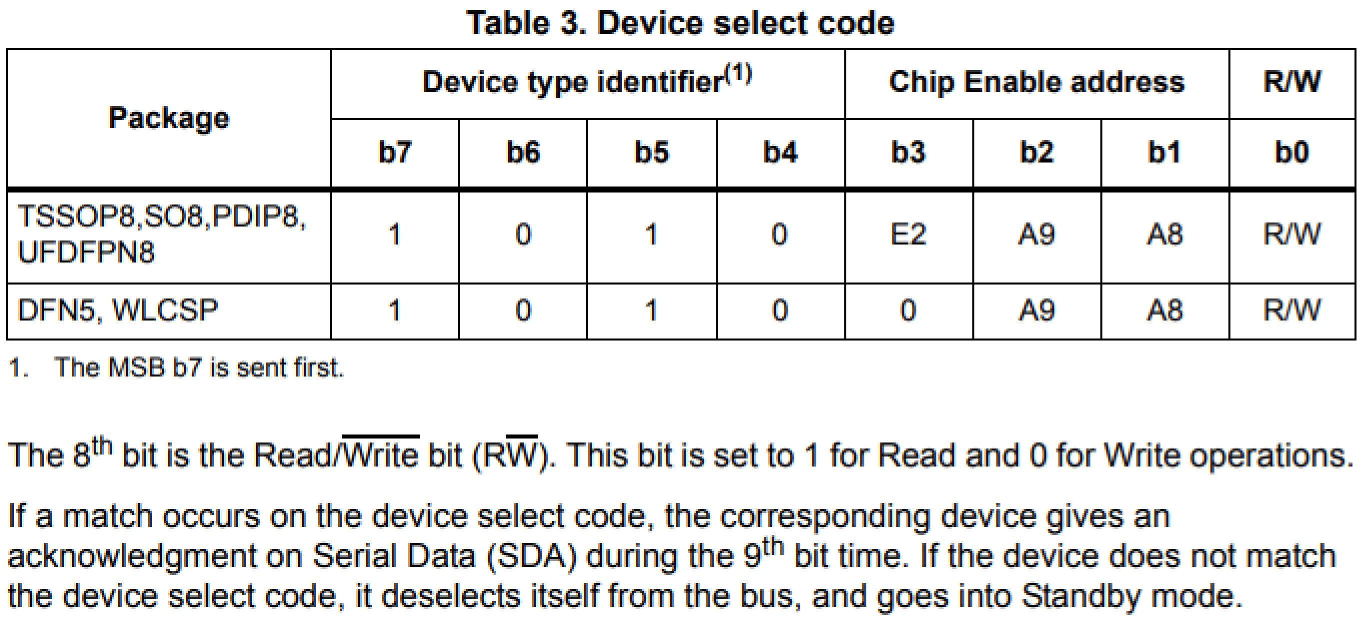


Table 3. Device select code

Package	Device type identifier(1)				Chip Enable address			R/W
	b7	b6	b5	b4	b3	b2	b1	b0
TSSOP8,SO8,PDIP8, UFDFPN8	1	0	1	0	E2	A9	A8	R/W
DFN5, WLCSP	1	0	1	0	0	A9	A8	R/W

1. The MSB b7 is sent first.

The 8th bit is the Read/$\overline{\text{Write}}$ bit (R$\overline{\text{W}}$). This bit is set to 1 for Read and 0 for Write operations.

If a match occurs on the device select code, the corresponding device gives an acknowledgment on Serial Data (SDA) during the 9th bit time. If the device does not match the device select code, it deselects itself from the bus, and goes into Standby mode.

AMD
together we advance_

Byte Write & Page Write

Dev Select = 1010100
如果要發送複數的資料時由低位開始發送，且每次發送的中間都要響應一次

Figure 8. Write mode sequences with $\overline{WC}$ = 0 (data write enabled)

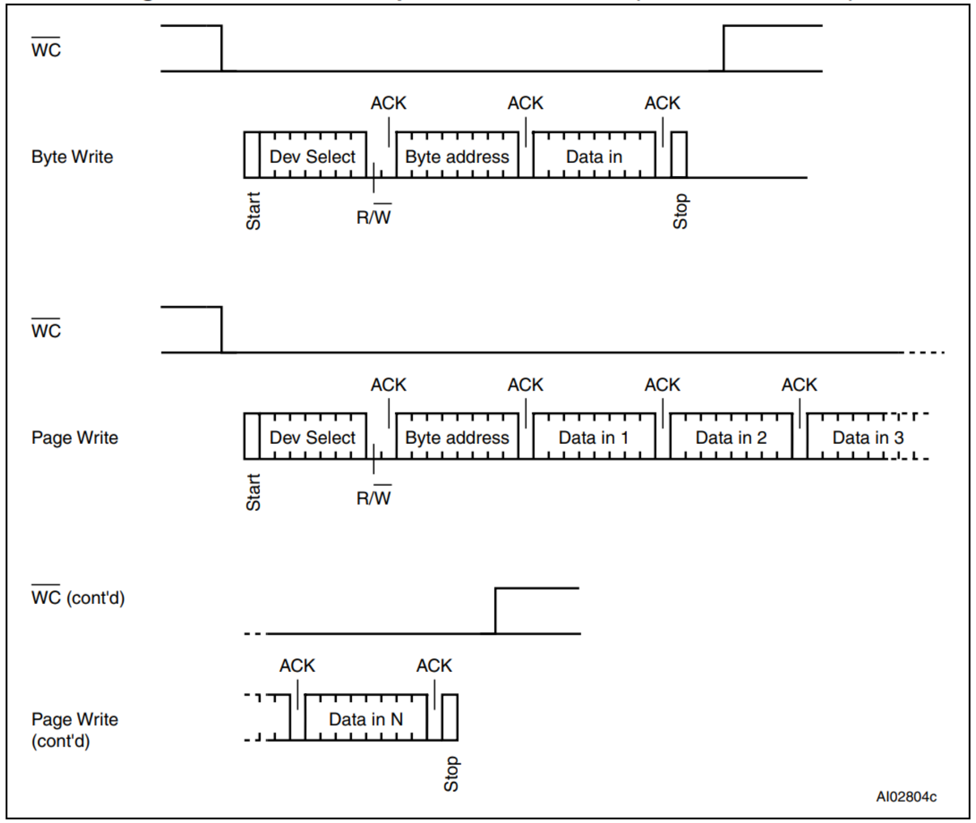
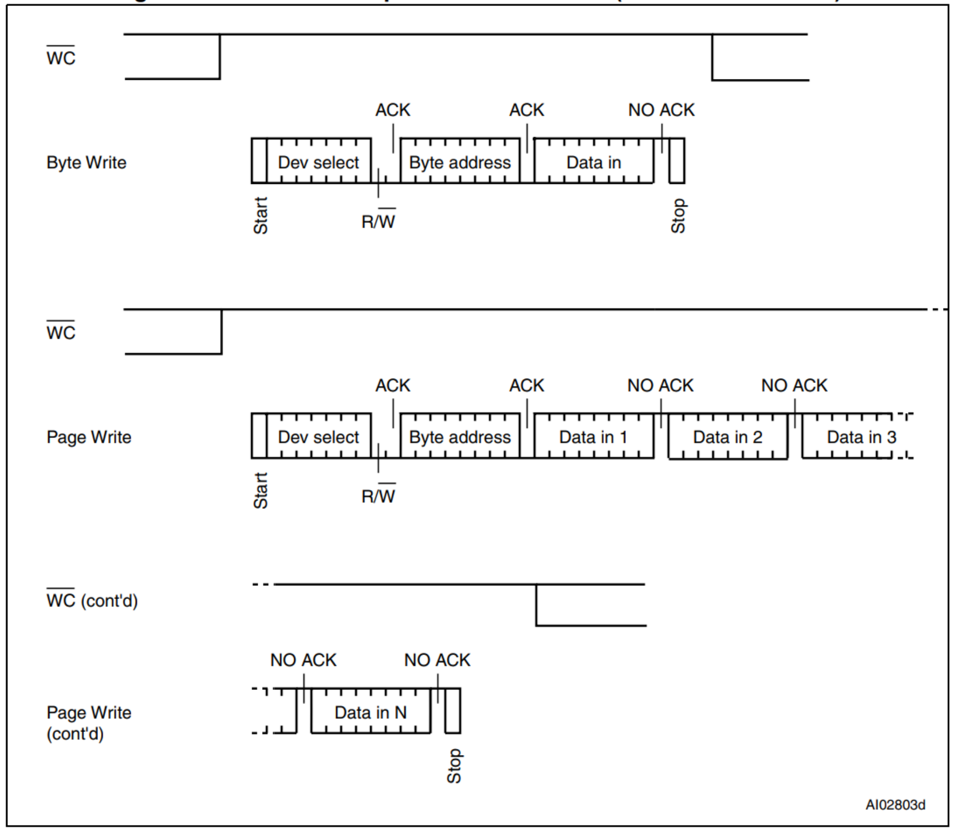


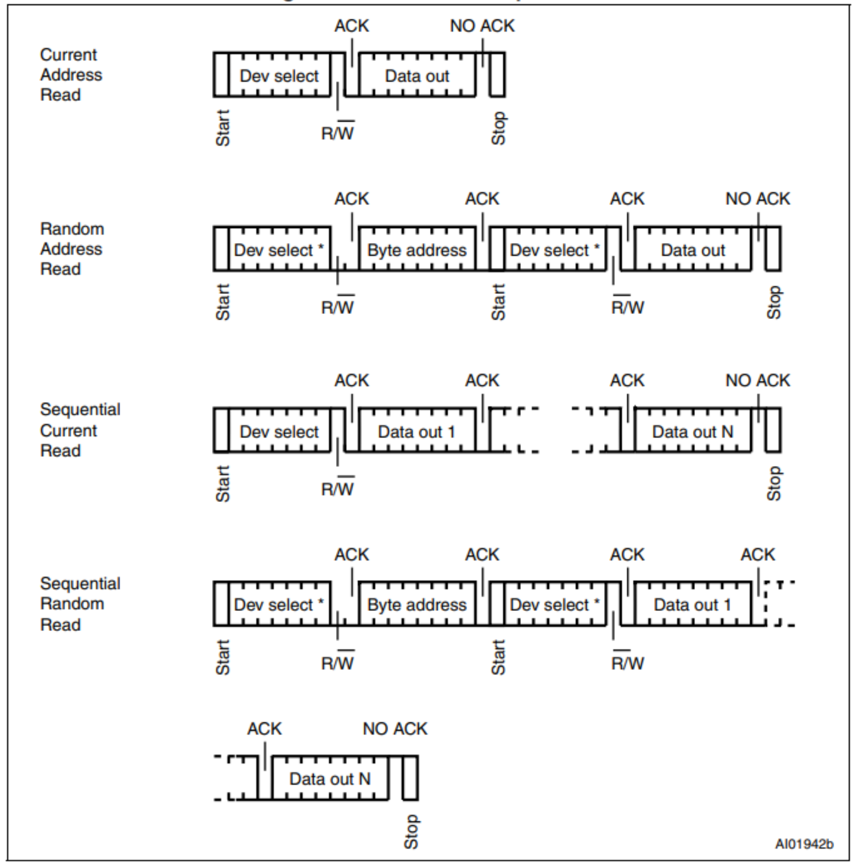
Figure 9. Write mode sequences with $\overline{WC}$ = 1 (data write inhibited)



Read operations

隨機讀可以理解為單Byte的讀取，另外因為主從機需要ID辨識，所以不管是寫入還是讀取都需要先enable一次寫入將從機ID寫入。讀取完成後拉低響應再發送停止位。

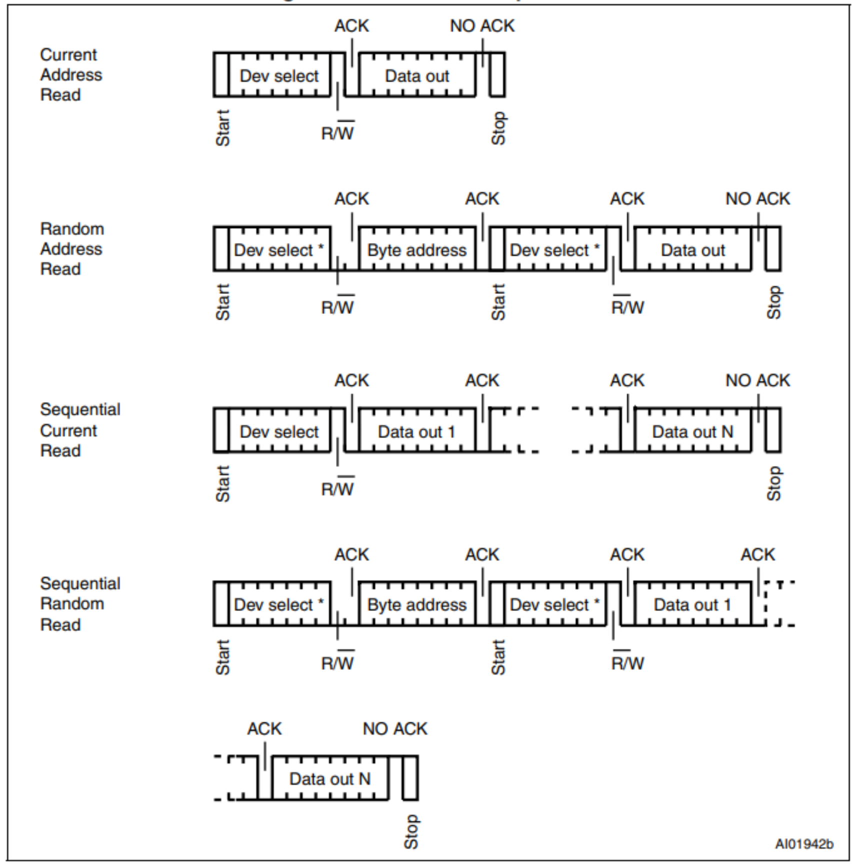
Figure 11. Read mode sequences



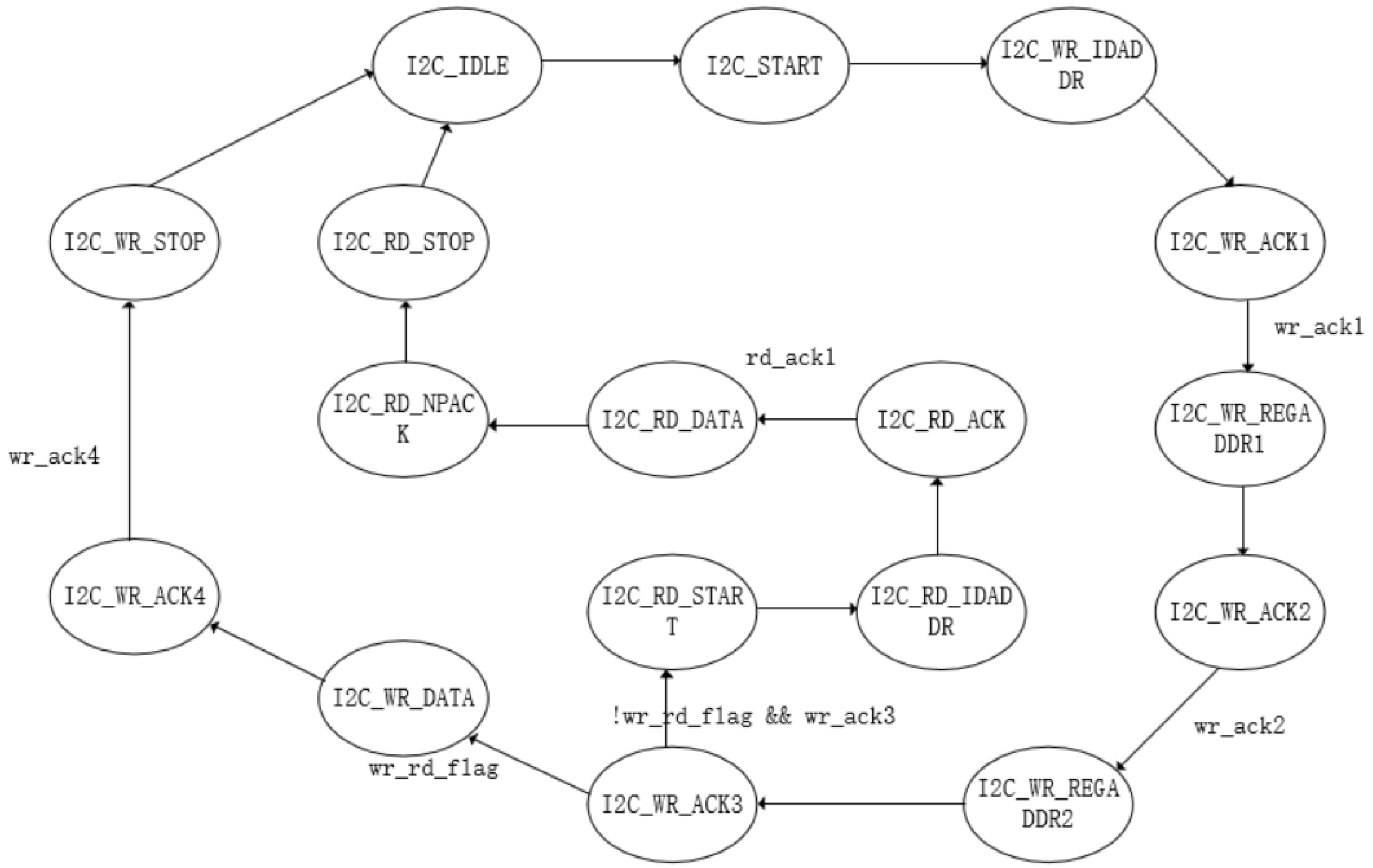
Read operations

隨機讀可以理解為單Byte的讀取，另外因為主從機需要ID辨識，所以不管是寫入還是讀取都需要先enable一次寫入將從機ID寫入。讀取完成後拉低響應再發送停止位。

Figure 11. Read mode sequences



FSM


I2C_IDLE
I2C_START
I2C_WR_IDADDR
I2C_WR_ACK1
wr_ack1
I2C_WR_REGADDR1
I2C_WR_ACK2
wr_ack2
I2C_WR_REGADDR2
I2C_WR_ACK3
!wr_rd_flag && wr_ack3
wr_rd_flag
I2C_RD_START
I2C_RD_IDADDR
I2C_RD_ACK
rd_ack1
I2C_RD_DATA
I2C_RD_NPACK
I2C_RD_STOP
I2C_WR_DATA
I2C_WR_ACK4
wr_ack4
I2C_WR_STOP

I2C_Ctrl_EEPROM

```
module I2C_Ctrl_EEPROM(
    input                       clk,
    input                       rst_n,
    input               [31:0]  eeprom_config_data,
    input                       i2c_start,
    inout                       i2c_sdat,
    output                      i2c_sclk,
    output                      i2c_done,
    output          reg [7:0]   i2c_rd_data
);
```

```
parameter       I2C_IDLE        =   'd0;
parameter       I2C_START       =   'd1;
parameter       I2C_WR_IDADDR   =   'd2;
parameter       I2C_WR_ACK1     =   'd3;
parameter       I2C_WR_REGADDR1 =   'd4;
parameter       I2C_WR_ACK2     =   'd5;
parameter       I2C_WR_REGADDR2 =   'd6;
parameter       I2C_WR_ACK3     =   'd7;
parameter       I2C_WR_DATA     =   'd8;
parameter       I2C_WR_ACK4     =   'd9;
parameter       I2C_WR_STOP     =   'd10;
parameter       I2C_RD_START    =   'd11;
parameter       I2C_RD_IDADDR   =   'd12;
parameter       I2C_RD_ACK      =   'd13;
parameter       I2C_RD_DATA     =   'd14;
parameter       I2C_RD_NPACK    =   'd15;
parameter       I2C_RD_STOP     =   'd16;
parameter       I2C_FREQ        =   250;
parameter       TRANSFER        =   1;
parameter       CAPTURE         =   125;
parameter       SEND_BIT        =   8;
```

[Public]

I2C_Ctrl_EEPROM

```
reg     [4:0]   pre_state;
reg     [4:0]   next_state;
//
reg             i2c_sdat_r;
wire            bir_en;
//
wire            transfer_en;
wire            capture_en;
reg             i2c_sclk_r;
reg     [7:0]   sclk_cnt;
//
reg     [3:0]   tran_cnt;
//
```

```
wire    [7:0]   wr_device_addr =
{eeprom_config_data[31:25], 1'b0};
wire    [7:0]   rd_device_addr =
{eeprom_config_data[31:25], 1'b1};
wire            wr_rd_flag  =   eeprom_config_data[24];
wire    [7:0]   reg_addr1   = eeprom_config_data[23:16];
wire    [7:0]   reg_addr2   = eeprom_config_data[15:8];
wire    [7:0]   wr_data     = eeprom_config_data[7:0];
//
reg             wr_ack1;
reg             wr_ack2;
reg             wr_ack3;
reg             wr_ack4;
reg             rd_ack1;
```

I2C_Ctrl_EEPROM

```verilog
//i2c_sclk
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
        sclk_cnt <= 'd1;
    else if(sclk_cnt == I2C_FREQ - 1'b1)
        sclk_cnt <= 'd0;
    else
        sclk_cnt <= sclk_cnt + 1'b1;
end
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
        i2c_sclk_r <= 1'b0;
    else if(sclk_cnt >= (I2C_FREQ>>2)*1 && sclk_cnt <= (I2C_FREQ>>2)*3)
        i2c_sclk_r <= 1'b1;
    else
        i2c_sclk_r <= 1'b0;
end
assign  transfer_en = (sclk_cnt == TRANSFER - 1)? 1'b1: 1'b0;
assign  capture_en  = (sclk_cnt == CAPTURE - 1)? 1'b1: 1'b0;
```

I2C_Ctrl_EEPROM

```
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
        tran_cnt <= 'd0;
    else if(tran_cnt == SEND_BIT && transfer_en == 1'b1)
        tran_cnt <= 'd0;
    else if(((next_state == I2C_WR_IDADDR || next_state == I2C_WR_REGADDR1 ||
        next_state ==I2C_WR_REGADDR2 || next_state == I2C_WR_DATA ||
        next_state == I2C_RD_IDADDR) && transfer_en == 1'b1) ||
        (next_state == I2C_RD_DATA && capture_en == 1'b1))
        tran_cnt <= tran_cnt + 1'b1;
    else
        tran_cnt <= tran_cnt;
end
```

I2C_Ctrl_EEPROM

```
//FSM step1
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
        pre_state <= I2C_IDLE;
    else
        pre_state <= next_state;
end
```

I2C_Ctrl_EEPROM

```
//FSM step2
always @(*)begin
    next_state = I2C_IDLE;
    case(pre_state)
    I2C_IDLE:
        if(i2c_start == 1'b1 && transfer_en == 1'b1)
            next_state = I2C_START;
        else
            next_state = I2C_IDLE;
    I2C_START:
        if(transfer_en == 1'b1)
            next_state = I2C_WR_IDADDR;
        else
            next_state = I2C_START;
    I2C_WR_IDADDR:
        if(transfer_en == 1'b1 && tran_cnt == SEND_BIT)
            next_state = I2C_WR_ACK1;
        else
            next_state = I2C_WR_IDADDR;
```

[Public]

I2C_Ctrl_EEPROM

```
    I2C_WR_ACK1:
        if(transfer_en == 1'b1 && wr_ack1 == 1'b0)
            next_state = I2C_WR_REGADDR1;
        else if(transfer_en == 1'b1)
            next_state = I2C_IDLE;
        else
            next_state = I2C_WR_ACK1;
    I2C_WR_REGADDR1:
        if(transfer_en == 1'b1 && tran_cnt == SEND_BIT)
            next_state = I2C_WR_ACK2;
        else
            next_state = I2C_WR_REGADDR1;
    I2C_WR_ACK2:
        if(transfer_en == 1'b1 && wr_ack2 == 1'b0)
            next_state = I2C_WR_REGADDR2;
        else if(transfer_en == 1'b1)
            next_state = I2C_IDLE;
        else
            next_state = I2C_WR_ACK2;
```

AMD
together we advance_

I2C_Ctrl_EEPROM

```
    I2C_WR_REGADDR2:
        if(transfer_en == 1'b1 && tran_cnt == SEND_BIT)
            next_state = I2C_WR_ACK3;
        else
            next_state = I2C_WR_REGADDR2;
    I2C_WR_ACK3:
        if(transfer_en == 1'b1 && wr_ack3 == 1'b0 && wr_rd_flag == 1'b0)
            next_state = I2C_WR_DATA;
        else if(transfer_en == 1'b1 && wr_ack3 == 1'b0 && wr_rd_flag == 1'b1)
            next_state = I2C_RD_START;
        else if(transfer_en == 1'b1)
            next_state = I2C_IDLE;
        else
            next_state = I2C_WR_ACK3;
    I2C_WR_DATA:
        if(transfer_en == 1'b1 && tran_cnt == SEND_BIT)
            next_state = I2C_WR_ACK4;
        else
            next_state = I2C_WR_DATA;
```

I2C_Ctrl_EEPROM

```
    I2C_WR_ACK4:
        if(transfer_en == 1'b1 && wr_ack4 == 1'b0)
            next_state = I2C_WR_STOP;
        else if(transfer_en == 1'b1)
            next_state = I2C_IDLE;
        else
            next_state = I2C_WR_ACK4;
    I2C_WR_STOP:
        if(transfer_en == 1'b1)
            next_state = I2C_IDLE;
        else
            next_state = I2C_WR_STOP;
    I2C_RD_START:
        if(transfer_en == 1'b1)
            next_state = I2C_RD_IDADDR;
        else
            next_state = I2C_RD_START;
```

I2C_Ctrl_EEPROM

```
    I2C_RD_IDADDR:
        if(transfer_en == 1'b1 && tran_cnt == SEND_BIT)
            next_state = I2C_RD_ACK;
        else
            next_state = I2C_RD_IDADDR;
    I2C_RD_ACK:
        if(transfer_en == 1'b1 && rd_ack1 == 1'b0)
            next_state = I2C_RD_DATA;
        else if(transfer_en == 1'b1)
            next_state = I2C_IDLE;
        else
            next_state = I2C_RD_ACK;
    I2C_RD_DATA:
        if(transfer_en == 1'b1 && tran_cnt == SEND_BIT)
            next_state = I2C_RD_NPACK;
        else
            next_state = I2C_RD_DATA;
```

I2C_Ctrl_EEPROM

```
    I2C_RD_NPACK:
        if(transfer_en == 1'b1)
            next_state = I2C_RD_STOP;
        else
            next_state = I2C_RD_NPACK;
    I2C_RD_STOP:
        if(transfer_en == 1'b1)
            next_state = I2C_IDLE;
        else
            next_state = I2C_RD_STOP;
    default:next_state = I2C_IDLE;
    endcase
end
```

I2C_Ctrl_EEPROM

```
//FSM step3
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)
        i2c_sdat_r <= 1'b1;
    else begin
        case(next_state)
        I2C_IDLE:   if(capture_en == 1'b1)  i2c_sdat_r <= 1'b1;
        I2C_START:  if(capture_en == 1'b1)  i2c_sdat_r <= 1'b0;
        I2C_WR_IDADDR:  if(transfer_en == 1'b1) i2c_sdat_r <= wr_device_addr['d7 - tran_cnt];
        I2C_WR_REGADDR1:if(transfer_en == 1'b1) i2c_sdat_r <= reg_addr1['d7 - tran_cnt];
        I2C_WR_REGADDR2:if(transfer_en == 1'b1) i2c_sdat_r <= reg_addr2['d7 - tran_cnt];
        I2C_WR_DATA:    if(transfer_en == 1'b1) i2c_sdat_r <= wr_data['d7 - tran_cnt];
        I2C_WR_ACK4:    if(transfer_en == 1'b1) i2c_sdat_r <= 1'b0;
        I2C_WR_STOP:    if(capture_en == 1'b1) i2c_sdat_r <= 1'b1;
        I2C_RD_START:   if(capture_en == 1'b1)  i2c_sdat_r <= 1'b0;
        I2C_RD_IDADDR:  if(transfer_en == 1'b1) i2c_sdat_r <= rd_device_addr['d7 - tran_cnt];
        I2C_RD_NPACK:   if(transfer_en == 1'b1) i2c_sdat_r <= 1'b0;
        I2C_RD_STOP:    if(capture_en == 1'b1) i2c_sdat_r <= 1'b1;
        default:        i2c_sdat_r <= i2c_sdat_r;
        endcase
    end
end
```

I2C_Ctrl_EEPROM

```
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        i2c_rd_data <= 8'b0;
        wr_ack1   <= 1'b1;
        wr_ack2   <= 1'b1;
        wr_ack3   <= 1'b1;
        wr_ack4   <= 1'b1;
        rd_ack1   <= 1'b1;
    end
    else if(capture_en == 1'b1)begin
        case(next_state)
        I2C_WR_ACK1: wr_ack1 <= i2c_sdat;
        I2C_WR_ACK2: wr_ack2 <= i2c_sdat;
        I2C_WR_ACK3: wr_ack3 <= i2c_sdat;
        I2C_WR_ACK4: wr_ack4 <= i2c_sdat;
        I2C_WR_STOP: begin
            wr_ack1   <= 1'b1;
            wr_ack2   <= 1'b1;
            wr_ack3   <= 1'b1;
            wr_ack4   <= 1'b1;
            rd_ack1   <= 1'b1;
        end
```

```
I2C_RD_ACK: rd_ack1 <= i2c_sdat;
        I2C_RD_DATA: i2c_rd_data['d7 - tran_cnt] <= i2c_sdat;
        I2C_RD_STOP:begin
            wr_ack1   <= 1'b1;
            wr_ack2   <= 1'b1;
            wr_ack3   <= 1'b1;
            wr_ack4   <= 1'b1;
            rd_ack1   <= 1'b1;
        end
        default:begin
            i2c_rd_data <= i2c_rd_data;
            wr_ack1 <= wr_ack1;
            wr_ack2 <= wr_ack2;
            wr_ack3 <= wr_ack3;
            wr_ack4 <= wr_ack4;
            rd_ack1 <= rd_ack1;
        end
        endcase
    end
    else begin
        i2c_rd_data <= i2c_rd_data;
        wr_ack1   <= wr_ack1;
        wr_ack2   <= wr_ack2;
        wr_ack3   <= wr_ack3;
        wr_ack4   <= wr_ack4;
        rd_ack1   <= rd_ack1;
    end
end
```

I2C_Ctrl_EEPROM

```
assign  bir_en = (pre_state == I2C_WR_ACK1 || pre_state == I2C_WR_ACK2 || pre_state == I2C_WR_ACK3 ||
                  pre_state == I2C_WR_ACK4 || pre_state == I2C_RD_ACK || pre_state == I2C_RD_DATA)? 1'b0: 1'b1;

assign  i2c_sdat = (bir_en == 1'b1)? i2c_sdat_r: 1'bz;

assign  i2c_sclk = i2c_sclk_r;
assign  i2c_done = (pre_state == I2C_WR_STOP && next_state == I2C_IDLE ||
                    pre_state == I2C_RD_STOP && next_state == I2C_IDLE)? 1'b1: 1'b0;

endmodule
```