Killing Two Birds with One Stone:

Learning to Write Better Verilog While Learning About the SPI Bus.

And Error Correction
(Time allowing)

```
//-------------------------------------------------
// Design Name : decoder_using_case
// File Name   : decoder_using_case.v
// Function    : decoder using case
// Coder       : Deepak Kumar Tala (minor changes by Mark Brehob)
//-------------------------------------------------
module decoder_using_case (
binary_in    , //  3-bit binary input
decoder_out , //  8-bit  out
enable          //  Enable for the decoder
);
input [3:0] binary_in  ;
input  enable ;
output [7:0] decoder_out ;

reg [7:0] decoder_out ;

always @ (enable or binary_in)
begin
  decoder_out = 0;
  if (enable) begin
    case (binary_in)
      3'h0 : decoder_out = 8'h01;
      3'h1 : decoder_out = 8'h02;
      3'h2 : decoder_out = 8'h04;
      3'h3 : decoder_out = 8'h08;
      3'h4 : decoder_out = 8'h10;
      3'h5 : decoder_out = 8'h20;
      3'h6 : decoder_out = 8'h40;
      3'h7 : decoder_out = 8'h80;

    endcase
  end
end

endmodule
```

```
1  //-------------------------------------------------
2  // Design Name : decoder_using_assign
3  // File Name   : decoder_using_assign.v
4  // Function    : decoder using assign
5  // Coder       : Deepak Kumar Tala (Minor changes by Mark Brehob)
6  //-------------------------------------------------
7  module decoder_using_assign (
8  binary_in    , //  3-bit binary input
9  decoder_out , //  8-bit out
10 enable          //  Enable for the decoder
11 );
12 input [2:0] binary_in  ;
13 input  enable ;
14 output [7:0] decoder_out ;
15
16 wire [7:0] decoder_out ;
17
18 assign decoder_out = (enable) ? (1 << binary_in) : 8'b0 ;
19
20 endmodule
```

```
1  //------------------------------------------------
2  // Design Name:   3 to 8 decoder
3  // Coder:         Deepak Kumar Tala (changes by Mark Brehob)
4  //------------------------------------------------
5  module decoder_using_assign (
6         input  [2:0]    binary_in,
7         input           enable,
8         output [7:0]    decoder_out
9  );
10
11       assign decoder_out = (enable) ? (1 << binary_in) : 16'b0 ;
12
13 endmodule
```

Option 1:

- `decoder_using_assign inst1(in, enable, out);`

Option 2a:

- `decoder_using_assign inst1(.binary_in(.in), .enable(enable), .decoder_out(out));`

Option 2b:

- ```
  decoder_using_assign inst1(
      .binary_in    (in     ),  //input [2:0]
      .enable       (enable ),  //input
      .decoder_out  (out    )   //output [7:0]
  );
  ```
  ```

```
 1 module sync_and_edges (
 2      input              clk,
 3      input              async_in,
 4      output             sync_out,
 5      output             rising_edge,                ,
 6      output             falling_edge
 7 );
 8      reg      [2:0]     tmp;
 9
10      always@(posreg clk)
11               tmp  <=  {tmp[1:0],async_in};
12
13      assign rising_edge        =tmp[2:1]==2'b01;
14      assign falling_edge       =tmp[2:1]==2'b10;
15      assign sync_cout          =tmp[1];
16 endmodule
```

So what does this code do?

```
always@(posreg clk)
          tmp  <=  {tmp[1:0],async_in};
```

- It's just a shift register.

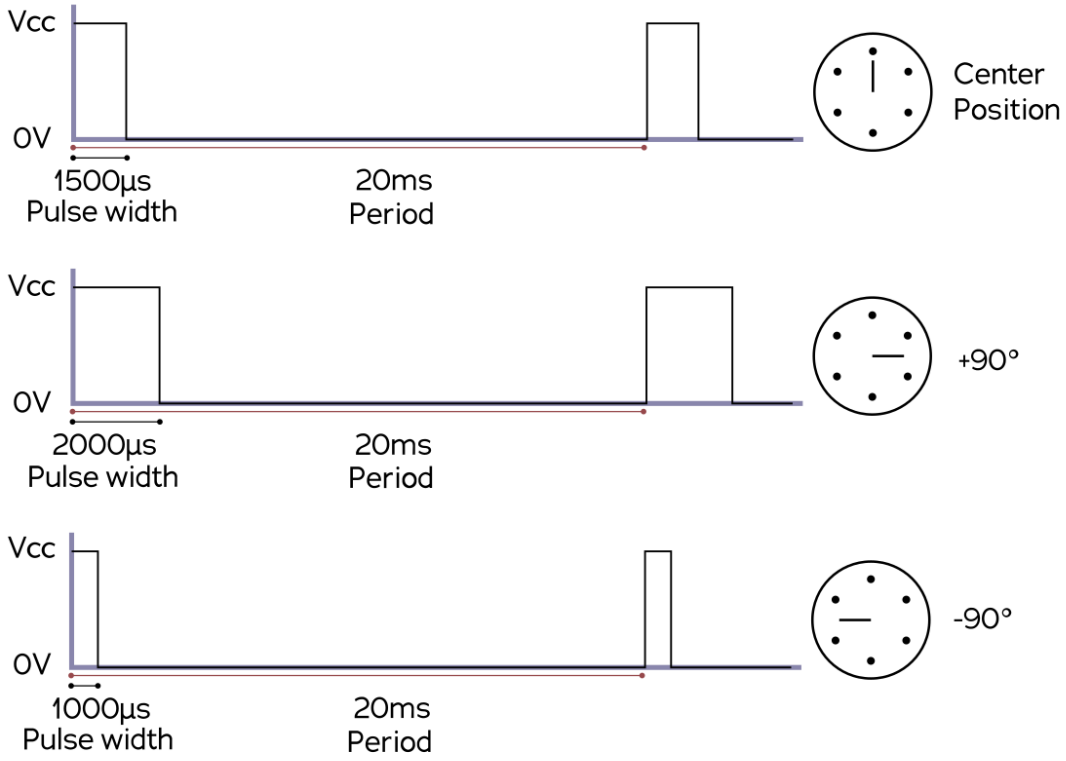
```
always@(posreg clk) begin
        tmp[2] <= tmp[1];
        tmp[1] <= tmp[0];
        tmp[0] <= async_in;
end
```

Serial communication

- When working with an embedded system, one key thing is communicating with external devices:
  - Sensors
  - Actuators (e.g. motors)
  - Displays
- Some devices have their own unique protocols.
  - Servos are a common example
    - You control them via pulse-width modulation
      - Basically a fixed period with a variable duty cycle.

Example: Servos

Their own fairly unique protocol

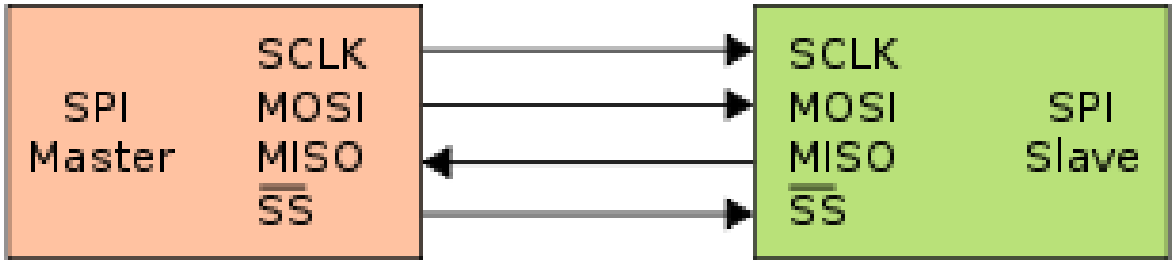


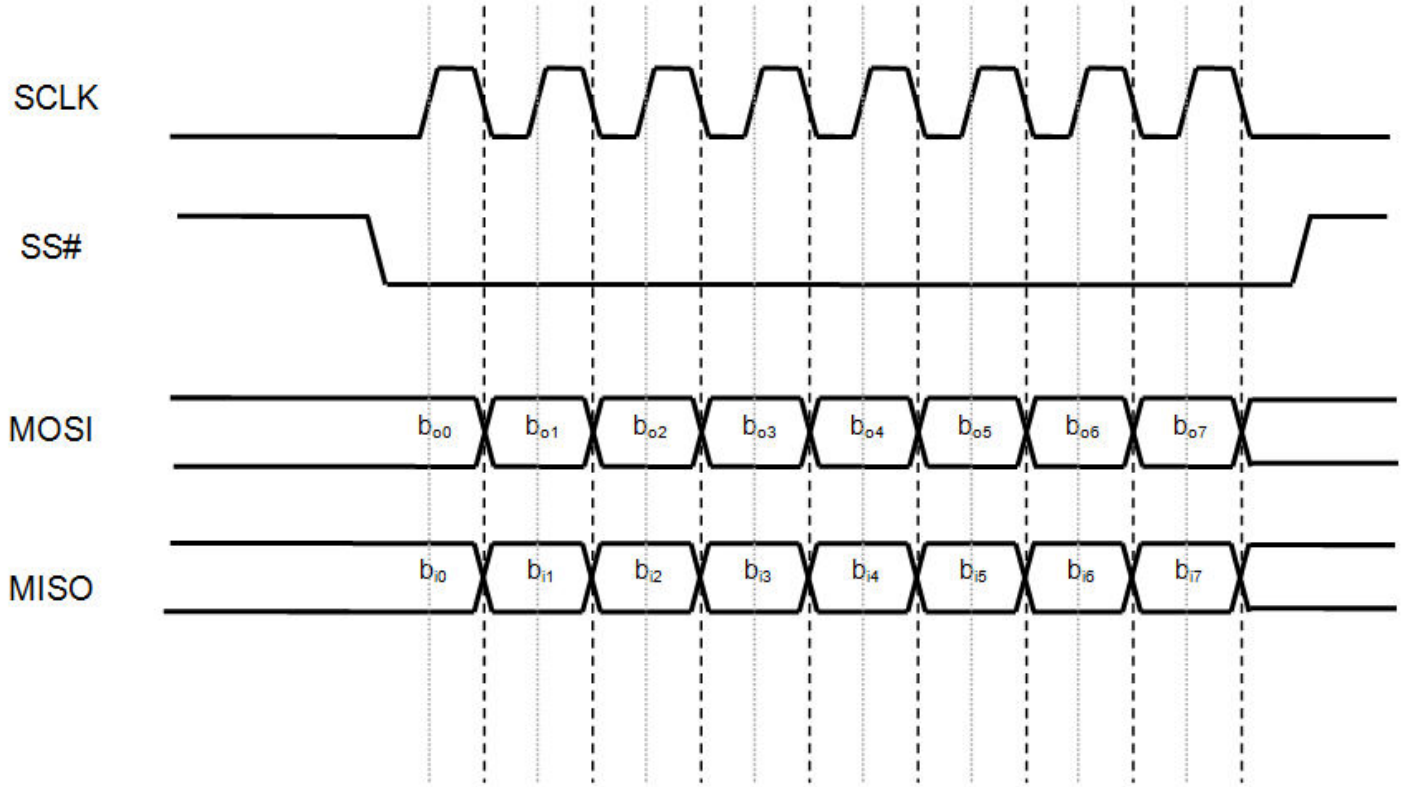
Standard interfaces

- There are three fairly standard interfaces in embedded systems
  - And a handful of others that see some use
- UART
  - Old protocol that was standard on PCs for years.
  - Most embedded systems that use USB really use UART and have a USB-to-UART converter chip.
- I2C
  - Fairly advanced protocol.
  - Addressable—each device has its own address and you can connect (in theory) dozens of devices to the same bus.

SPI—Serial Peripheral Interface

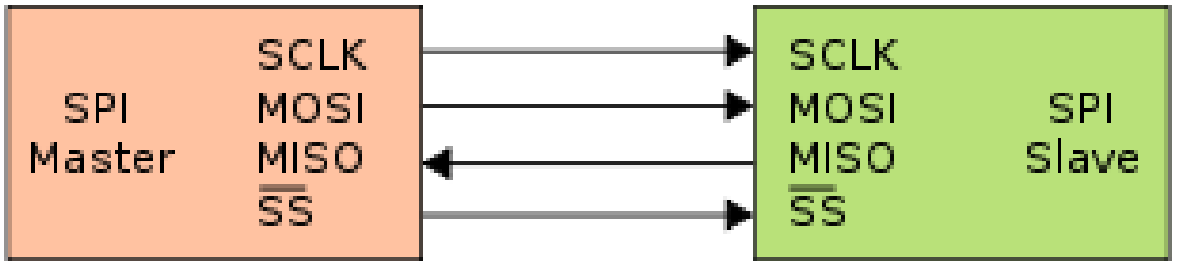
- 4 wires:
  - Clock
  - Data from master to servant
  - Data from servant to master
  - Select
- Runs in full duplex
  - Data goes both ways at the same time.
- Not really a standard, more of a hack.
  - So lots of weird issues to deal with as different people build slightly different interfaces…


SPI
Master
SCLK
MOSI
MISO
SS
SCLK
MOSI
MISO
SS
SPI
Slave


SCLK
SS#
MOSI
MISO
b_{o0} b_{o1} b_{o2} b_{o3} b_{o4} b_{o5} b_{o6} b_{o7}
b_{i0} b_{i1} b_{i2} b_{i3} b_{i4} b_{i5} b_{i6} b_{i7}

What do we need to do?

Interface




```
1 module SPI_servant(
2     input                   clk,
3     input                   SCK,
4     input                   MOSI,
5     input                   SSEL,
6     input           [7:0]   SSW,
7     output                  MISO,
8     output reg      [7:0]   SLED
9 );
```

Synchronize and edges

```
//*****************************************************************
// Get 3 versions of each SPI input:
// Syncronized, rising edge, and falling edge.
    wire SCK_s,  SCK_re,  SCK_fe;
    wire MOSI_s, MOSI_re, MOSI_fe;
    wire SSEL_s, SSEL_re, SSEL_fe;

    sync_and_edges SCK_inst (clk, SCK,  SCK_s,  SCK_re,  SCK_fe );
    sync_and_edges SSEL_inst(clk, SSEL, SSEL_s, SSEL_re, SSEL_fe);
    sync_and_edges MOSI_inst(clk, MOSI, MOSI_s, MOSI_re, MOSI_fe);
```

```
 1 module sync_and_edges (
 2     input             clk,
 3     input             async_in,
 4     output            sync_out,
 5     output            rising_edge,              ,
 6     output            falling_edge
 7 );
 8     reg      [2:0]    tmp;
 9
10     always@(posreg clk)
11             tmp  <=  {tmp[1:0],async_in};
12
13     assign rising_edge        =tmp[2:1]==2'b01;
14     assign falling_edge       =tmp[2:1]==2'b10;
15     assign sync_cout          =tmp[1];
16 endmodule
```

Getting data

```
24 //*************************************************************
25 // Shift data in.  Could use a counter, but will assume
26 // the input is well-formed (SSEL low then 8 clocks then SSEL high)
27
28     reg    [7:0]    data_received;
29     always@(posedge clk)
30     begin
31         if(SSEL_s)
32             data_received<=data_received; // hold (just for clarity)
33         else if(SCK_re)
34                 data_received<={data_received[6:0],MOSI_s};
35
36         if(SSEL_re)
37             SLED<=data_received;  //once SS is high, data done.
38     end
```

Sending Data

```
40 //*************************************************************
41 // Send data out.  We set data on falling edge.  Need
42 // to be sure we have the right data at the start too.
43
44     reg    [7:0]    data_to_send;
45     assign MISO=SSEL_s?data_to_send[7]:1'bz;
46     always@(posedge clk)
47     begin
48         if(SSEL_fe)
49             data_to_send<=SSW; // grab data
50         if(SCK_fe)
51             data_to_send<={data_to_send[6:0],1'b0};  // shift on falling edge.
52     end
53
```

```
module SPI_servant(
    input                clk,
    input                SCK,
    input                MOSI,
    input                SSEL,
    input          [7:0] SSW,
    output               MISO,
    output reg     [7:0] SLED
);


//*********************************************************
// Get 3 versions of each SPI input:
// Syncronized, rising edge, and falling edge.
    wire SCK_s,  SCK_re,  SCK_fe;
    wire MOSI_s, MOSI_re, MOSI_fe;
    wire SSEL_s, SSEL_re, SSEL_fe;

    sync_and_edges SCK_inst (clk, SCK,  SCK_s,  SCK_re,  SCK_fe );
    sync_and_edges SSEL_inst(clk, SSEL, SSEL_s, SSEL_re, SSEL_fe);
    sync_and_edges MOSI_inst(clk, MOSI, MOSI_s, MOSI_re, MOSI_fe);


//*********************************************************
// Shift data in.  Could use a counter, but will assume
// the input is well-formed (SSEL low then 8 clocks then SSEL high)

    reg    [7:0]    data_received;
    always@(posedge clk)
    begin
        if(SSEL_s)
            data_received<=data_received; // hold (just for clarity)
        else if(SCK_re)
                data_received<={data_received[6:0],MOSI_s};

        if(SSEL_re)
            SLED<=data_received;  //once SS is high, data done.
    end

//*********************************************************
// Send data out.  We set data on falling edge.  Need
// to be sure we have the right data at the start too.

    reg    [7:0]    data_to_send;
    assign MISO=SSEL_s?data_to_send[7]:1'bz;
    always@(posedge clk)
    begin
        if(SSEL_fe)
            data_to_send<=SSW; // grab data
        if(SCK_fe)
            data_to_send<={data_to_send[6:0],1'b0};  // shift on falling edge.
    end

endmodule
```

One more application: Error correction

- Say we are storing data (or communicating with it)
  - Sometimes it goes bad.
    - Why?
- How can we recover it?
  - Have some redundancy!
- But keeping a copy of the data is too much.
  - I don’t want 100% overhead!

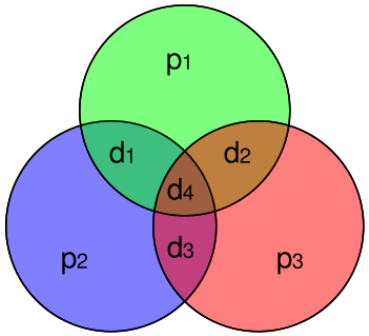


Figure from Wikipedia

Example block code: Hamming(7,4)

- Hamming(7,4)-code.
  - Take 4 data elements (d_1 to d_4)
  - Add 3 parity bits (p_1 to p_3)
    - $p_1=P(d_1, d_2, d_4)$
    - $p_2=P(d_1, d_3, d_4)$
    - $p_3=P(d_2, d_3, d_4)$
  - If any one bit goes bad (p or d) can figure out which one.
    - Just check which parity bits are wrong. That will tell you which bit went wrong.
    - If more than one went wrong, scheme fails.
- Much more efficient on larger blocks.
  - E.g. (136,128) code exists.
- Example:
  - Say
    - d[1:4]=4'b0011
  - Then:
    - $p_1=P(0,0,1)=1$
    - $p_2=P(0,1,1)=0$
    - $p_3=P(0,1,1)=0$
  - If d_2 goes bad (is 1)
    - Then received p_1 and p_3 are wrong.
    - Only d_3 covered by both (and only both)
      - ***So d_3 is the one that flipped.***