Serial Communication Through an Asynchronous FIFO Buffer

Final Project Report
December 9, 2000
E155

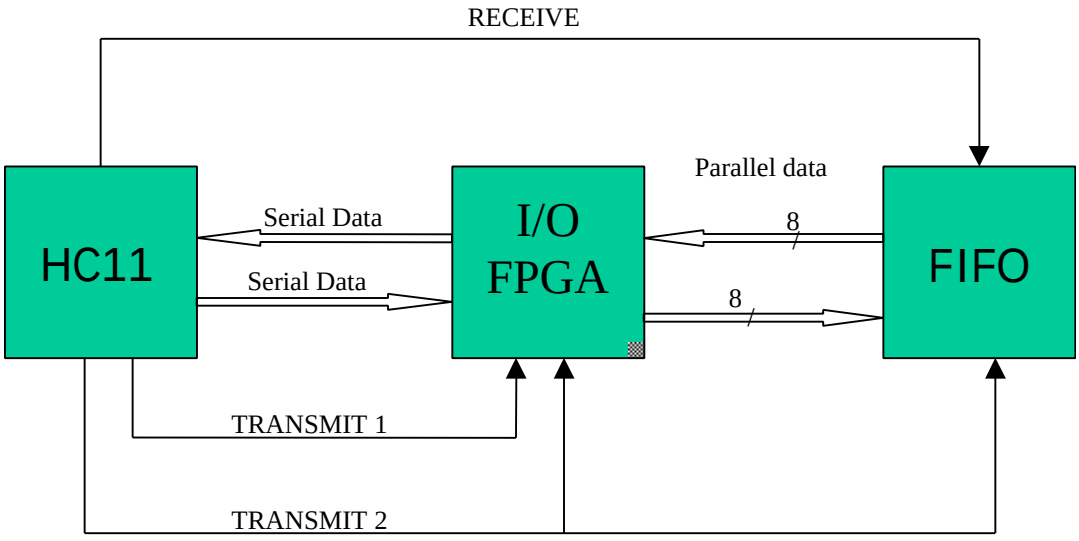
Nick Bodnaruk and Andrew Ingram

Abstract:

For our clinic, we need to be able to use serial communication with a PC to exchange data with an asynchronous FIFO Buffer on a PCB. In order to gain insight in this process, we set out to use the HC11 and two FPGAs, one acting as the FIFO and another as a translator, to send data through the FIFO and back to itself. The FIFO has four stages of memory, and is not controlled by a clock. The translator FPGA converts serial data to parallel and vice-versa. The HC11 successfully sent four eight bit words, but due to control problems, only received seven of the bits sent for each word.

Introduction

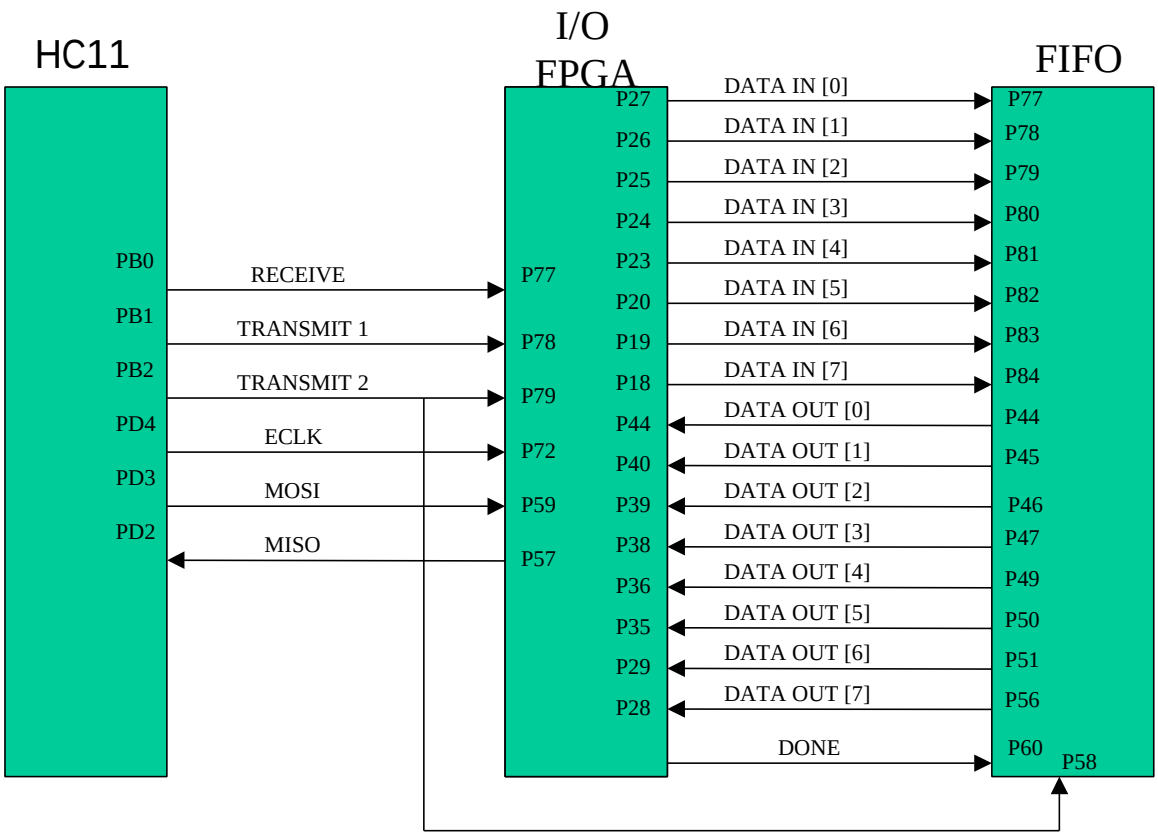
As members of the Sun Microsystems clinic team, we have been asked to design and fabricate a demonstration PCB that contains an asynchronous FIFO. It will need to be able to communicate with a PC via a serial connection, among other requirements. To gain experience with the intricacies of serial communication and how a FIFO works, we decided to use the HC11 microcontroller and two other FPGAs for to perform these functions. The HC11 sends serial data to a translator FPGA, which converts the data into parallel data, and sends it to the FIFO FPGA. To take data out of the FIFO, the translator takes the data from the FIFO's last stage, and sends it serially back to the HC11. A block diagram of our system is shown below.



The following sections describe how each component functions and how they were designed, as well as how the integration of everything was achieved. The HC11 used its Serial Peripheral Interface for communication and was controlled using assembly language. The FPGAs were programmed with a combination of schematics and verilog code.

Schematics

This is the overall design of our system, including pinouts and the name of each signal. Each signal and its purpose are described in the following sections.



Microcontroller Design

For our project, we used the Motorola MC68HC11 microcontroller in place of a PC equipped with a serial port for sending and receiving data to and from our FIFO. The implementation was fairly simple, although we did have one problem that we were unable to solve. We simply stored in memory the data elements that we wished to send to our FIFO. For ease of use in sending them out to the FIFO, they were stored in consecutive memory locations on the zero page on the HC11. This allowed us to simply use the X register as a pointer to which data item should be sent out over the built in serial peripheral interface (SPI) to the I/O FPGA. After each data item was sent out, we incremented X, which ensures that we would subsequently send the next data item in our set.

As mentioned above, data is written to and read from the I/O FPGA using the SPI. Data was sent out from the HC11 to the FPGA over the MOSI interface, and data was read back into the FPGA over the MISO line. We configured the SPI control register so that the HC11 was set as master, the serial clock (ECLK) idles high, and that the rising edge of ECLK was the active edge. We believe that ECLK idling high is partly responsible for problems we will discuss later. It is possible to set ECLK to idle low, but according to Cady, the SSbar (Slave Select) signal "must be deasserted and reasserted between each successive serial byte". We were unsure of any other implications of doing this, and so decided to have ECLK idle high.

Since the serial data registers in the HC11 and the slave can be thought of as one circular register, we must write data out of the serial port in order to read data from our FPGA. In order to prevent the FPGA from accepting this as valid data, we have another

signal (RECEIVE) that is sent out over the parallel port that alerts the FPGA that the data is not valid. Additionally, we have two other signals that are generated on the HC11: TRANSMIT 1 and TRANSMIT 2. TRANSMIT 1 causes the I/O FPGA to latch the data at the output of the FIFO. TRANSMIT 2 enables the shift register to shift the data back to the HC11, and it also alerts the FIFO control that we have removed a data item from the FIFO.

Data items were sent out four at a time to the I/O FPGA, where they were then sent to the FIFO. After all data was sent, the HC11 branches to the “sent” label of the code (see appendix C), and then retrieves all four of the data items consecutively. We have two delay loops in our code (hold and hold2) that ensure that the data transfers are finished before we try to begin another transfer.

Because we are interfacing with a FIFO, we expect to receive the same data back and in the same order that we sent it out in. Once we get the data back from the I/O FPGA, we store it in memory so that it can be compared to the data that we wrote to the FIFO.

FPGA Design

The control circuitry we are using was created by Sun Microsystems and is called the asP* protocol, which stands for Asynchronous Symmetric Persistent Pulse Protocol. A single control stage is shown in Appendix B, under the One_Stage_Control Verilog code. It is asynchronous because there is no clock input, and symmetric because when any number of stages are placed together, they are exactly the same in either direction (left to right and vice versa). Persistent pulse refers to the local handshake signals that convey information between each stage. When no data is being passed or received they stay at a constant logic level, but when an event occurs they pulse to the opposite logic level for a short amount of time. This does present a problem when attempting to place a single data item in the FIFO, and its resolution will be discussed shortly.

As can be seen from the FIFO FPGA in Appendix A, the FIFO we implemented has four stages. The local handshake signals are the connections between SEIn_ and SEOut_, and FIn and FOut. The F signal indicates whether the previous stage is full or not, while the SE signal pulses to change the previous full stage to empty, while also changing the next stage from empty to full. This way, the full signal travels until it hits either the end of the FIFO or another full stage. The MV signal is an empty indicator, and acts as the enable for the memory elements that hold and transfer the data, while also driving the LED empty indicators.

To place a single data item in the FIFO, there must be a way to indicate to the control circuitry that only one stage is becoming full. This is accomplished by sending a pulse to FIn of the first stage, which is created by the pulse generator PG, shown in Appendix B. When the input goes from LOW to HIGH, the output is initially LOW,

goes to HIGH for a short amount of time, and then returns to LOW. Notice the similarity between the pulse generator logic and a stage of the control circuitry. The reason for this is to create a pulse that is of the same duration as the control's handshake pulse. This ensures that only one stage will become full, since if the pulse was longer, the control would see this as more data being added.

The FIFO has a data depth of eight bits, and we used enabled SR latches as the memory elements. These latches are shown in the 3LATCH module in Appendix A. The IN signal shown in the schematic controls when the first stage of the FIFO goes from empty to full, while the OUT signal controls when the last stage goes from full to empty. The MV signals are sent to output pins to drive LEDs, to view which stages are empty. Data is presented to the FIFO at the DIN inputs, and the last stage's data is output at the DOUT pins.

The second FPGA we used (I/O FPGA, shown in Appendix A) acted as the translator between the FIFO and the HC11. Examining first the receiving aspect, the three control signals are ECLK, RECEIVE, and MOSI. ECLK is the serial clock that cycles eight times. When RECEIVE is high, the first shift register (SHIFTREG1) will shift in the serial data given by MOSI. Otherwise, the shift register will do nothing. This converts the serial data into parallel for the FIFO. Also, when half of the data is shifted in, a DONE signal is generated. The counter used for this is three bits, so that it will overflow back to zero when eight bits are shifted. This signal is input into the 6DELAY module, where it is delayed by six cycles of the I/O FPGA's clock (running at 1MHz). This delays the signal enough to ensure that the shifting is done before it tells the FIFO to make one stage full.

The other function of the I/O FPGA is to take the parallel data from the FIFO's last stage and send it serially to the HC11. This is accomplished through SHIFTREG2 (shown in appendix B), which is controlled by ECLK, TRANSMIT1, and TRANSMIT2. Since ECLK is active high, we took its inverse and combined it with TRANSMIT1 to form the clock input for the registers. TRANSMIT2 controls whether the registers will shift data serially or take data from the FIFO. With 2 low, sending 1 high causes data to be taken into the registers. Then, sending 2 high and 1 low, when ECLK cycles eight times the data will be shifted out through MISO to the HC11. The TRANSMIT2 signal is also the input to the FIFO OUT signal, causing the last stage to transition from full to empty.

Results

We sent four eight-bit data words to the FIFO, where they were stored successfully and the FIFO was filled up. However, when trying to take the data out of the FIFO and send it back to the HC11, the FIFO emptied correctly, but a single bit was lost in each word during the serial communication. This will be discussed shortly. With the exception of one data bit not being received, we accomplished our goals as presented in our project proposal, namely the construction of an asynchronous FIFO on an FPGA and communicating with it serially.

The first major problem we ran into was to actually implement an asynchronous FIFO such that it functioned correctly. It turned out that the FIFO is extremely sensitive to timing issues, such that the placement of logic determines if it will work or not. For example, just by using the floorplanner to move a few LUTs, we were able to break the FIFO. This is the main reason why we did not incorporate the FIFO and translator onto a single FPGA, since the FIFO did not function, even though the logic for the FIFO was placed in exactly the same positions.

The other problem, which we were unable to correct in time, was the loss of a data bit. We were trying to control one set of flip flops with two clock signals, one to latch the data, and one to shift it out serially. Doing this proved difficult, and ultimately not quite possible for us. We got close, but we ended up missing a data bit when it was shifted from the second I/O shift register. This resulted in $14 becoming $28, $03 becoming $06, and all other data items being shifted one too many times to the left. This was caused by the manner in which we tried to implement the two-clocked register: we tried to combine the two clocks into one clock that would then control the flip flops.

We did this by inverting the ECLK, and then ORing it with the parallel TRANSMIT1 signal that was generated by the HC11 (see Appendix B). However, when ECLK is inverted, this causes the positive edge to be moved forward by half a cycle. Thus, when we go to shift data back into the HC11, the first bit comes earlier than the HC11 is expecting it, and so this first bit is missed in the transfer. This in effect doubles the value of the data received, since the MSB is not received.

A possible solution that we were unable to implement would be to have TRANSMIT1 be high normally, and pulse low when latching data from the FIFO. Then, you would only need to AND TRANSMIT1 and ECLK together, thus preserving the positive edge of ECLK at the same position. Then, the data sent through MISO will occur at the expected time for the HC11, and all eight data bits would be received correctly.

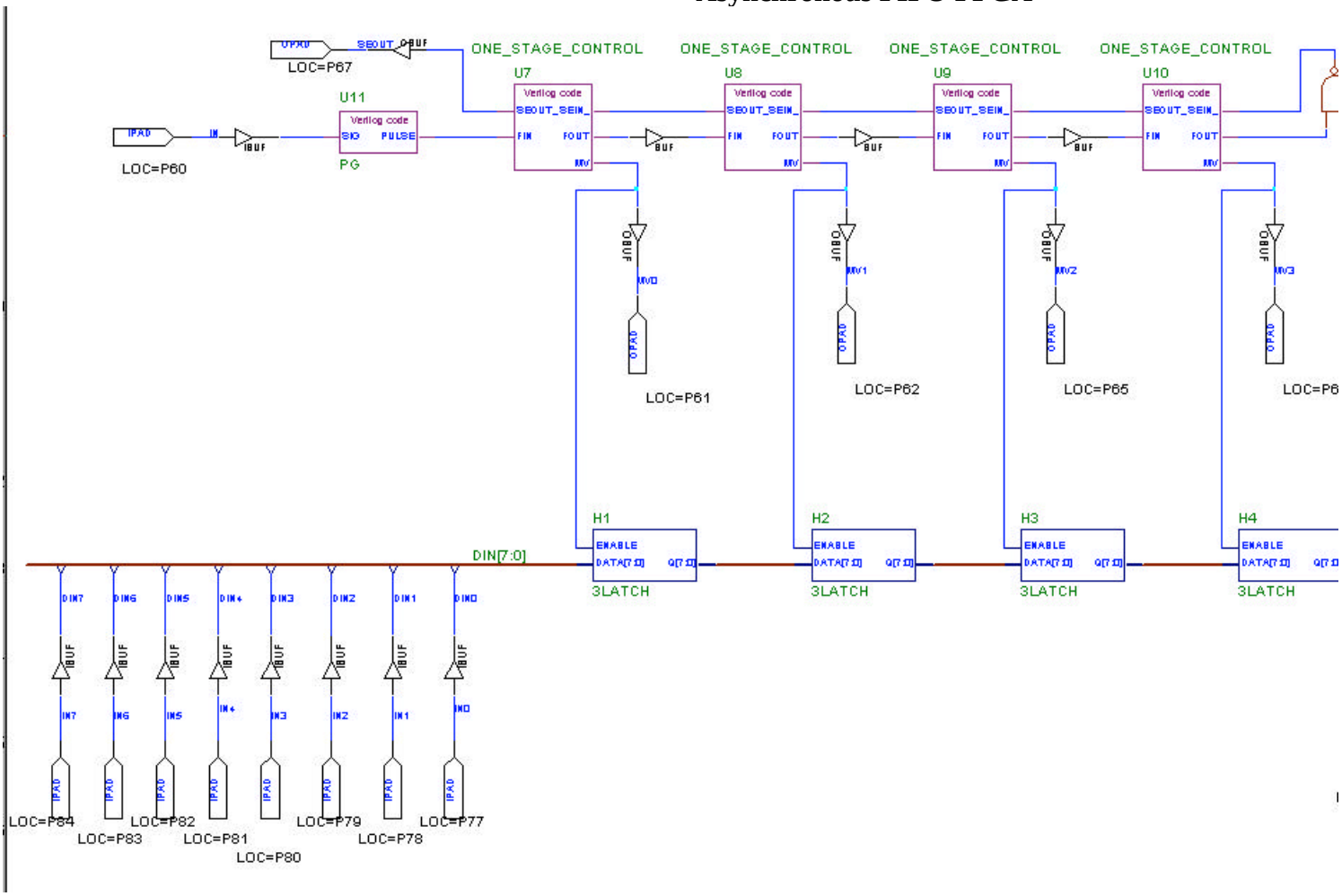
References

[1] Cady, Fredrick M. Software and Hardware Engineering: Motorla M68HC11, Oxford University Press, 1997.

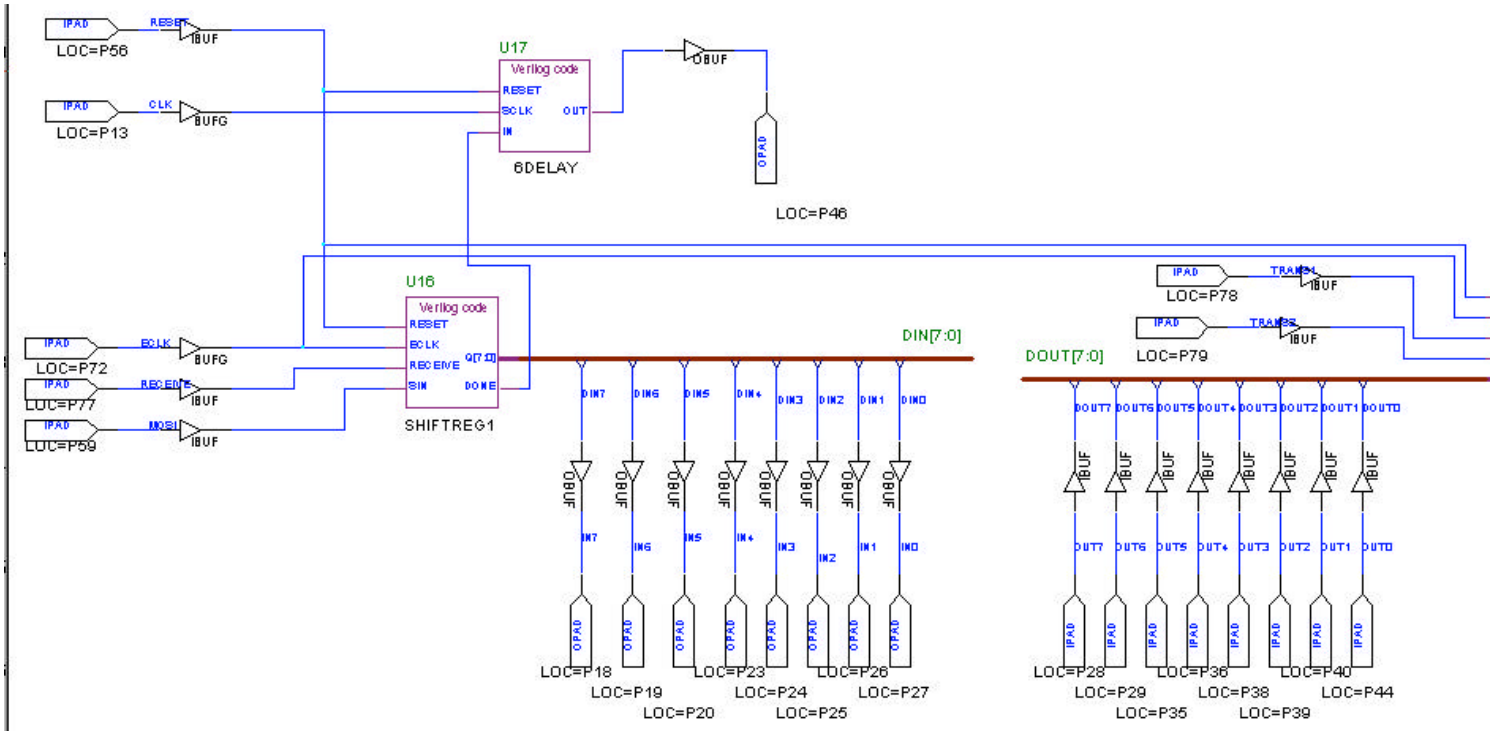
[2] Molnar, Charles E., et al. Two FIFO Ring Performance Experiments. Palo Alto: Sun Microsystems, Inc. 1999.

Appendix A: FPGA Schematics

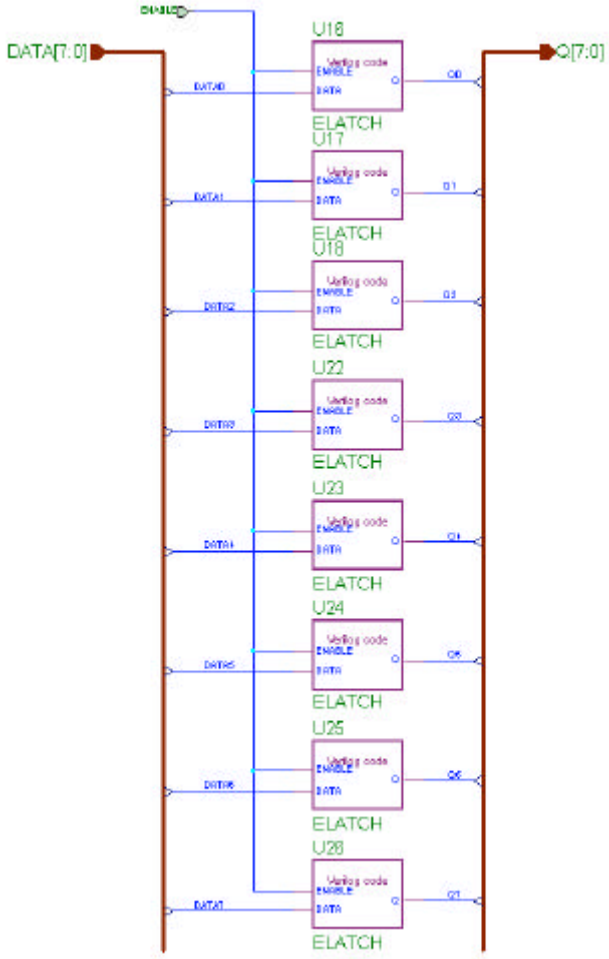
Asynchronous FIFO FPGA



In/Out (I/O) FPGA

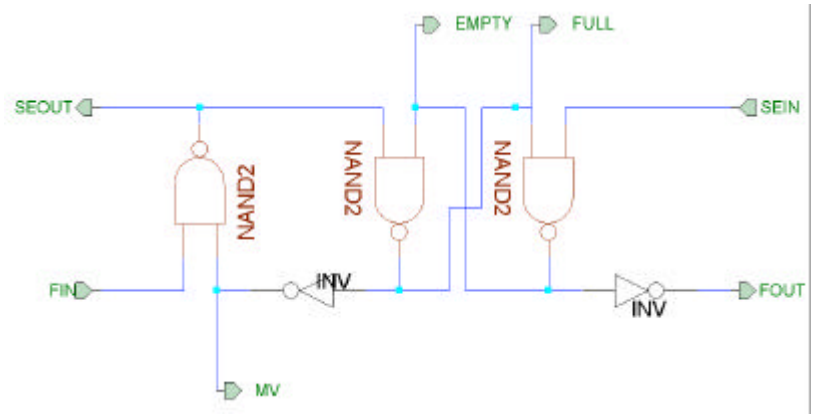


3LATCH Module



Appendix B: Verilog Code

```
module One_Stage_Control (FIn, FOut, SEOut_, SEIn_, MV) ;

// This module is a single stage of the FIFO control logic
// The F signals transmit the stage's full status, while the SE
// signals are the handshake pulses that act to change the FIFO
// fullness

input  FIn, SEIn_ ;
output FOut, SEOut_, MV;

wire   Full, Empty;

assign SEOut_ = ~(FIn & (~Full));
assign Empty = ~(Full & SEIn_);
assign FOut = ~Empty;
assign Full = ~(Empty & SEOut_);
assign MV = ~Full;

endmodule
```




```
module Pulse_Gen (SIG, PULSE) ;

// This module is a level to pulse converter.  When the incoming signal
// (SIG) goes high, the output pulses high for a small amount of time.
// Because the logic for this module is very similar to the control
// logic, the pulse is of the correct length to indicate only one
// data item is entering the FIFO

input  SIG ;
output PULSE ;

wire   Hand, E, F;

assign Hand = ~(SIG & ~(F));
assign F = ~(Hand & E);
assign E = ~(F & SIG);
assign PULSE = ~(Hand);

endmodule
```

```
module Elatch (Enable, Data, Q) ;

// This module creates a simple SR Latch using NAND gates

input  Enable ;
input  Data ;
output Q ;

wire Set, Reset, Qbar;

assign Set = ~(Enable & ~(Data));
assign Reset = ~(Enable & Data);
assign Qbar = ~(Set & Q);
assign Q = ~(Reset & Qbar);

endmodule

module shiftreg1 (eclk, reset, sin, receive, q, done) ;

// This module takes the serial data from the HC11, and converts it
// into a parallel output for the FIFO

input  eclk ;     // serial clock
input  reset ;
input  sin ;      // serial data in
input  receive ;  // selects if register is receiving
output [7:0] q ;
output done ;     // tells when the shift register is half full

reg    [2:0] count ;    // 3-bit count signal

reg    [7:0] q ;        // The shift register

always @(posedge eclk or posedge reset)
if (reset) count <= 3'b0;
else if (receive) begin
      q[0] <= sin;      // When receive is high, shift for every
      q[1] <= q[0];     // clock tick
      q[2] <= q[1];
      q[3] <= q[2];
      q[4] <= q[3];
      q[5] <= q[4];
      q[6] <= q[5];
      q[7] <= q[6];
      count <= count+1;
      end

assign done = count[2]; // done goes high when the shift register
                        // is half full
endmodule
```

```
module SixDelay (in, sclk, reset, out) ;

// This module causes a delay of six sclk cycles between in and out
// Its purpose is to make sure the FIFO does take the data before it
// is done being shifted into the register.

input in ;                    // input
input sclk ;                  // system clock
input reset ;
output out ;                  // output

reg [5:0] shift ;

always @(posedge sclk or posedge reset)
if (reset) shift <= 6'b0;
else begin
      shift[0] <= in;
      shift[1] <= shift[0];
      shift[2] <= shift[1];
      shift[3] <= shift[2];
      shift[4] <= shift[3];
      shift[5] <= shift[4];
      end

assign out = shift[5];

endmodule
```

```
module Shiftreg2 (eclk, reset, transmit1, transmit2, last, sout) ;

// This module converts the parallel output of the FIFO to the serial
// input needed for the HC11.  Two transmit control signals are needed,
// one to tell the registers to take data from the FIFO, and another
// to enable the serial shifting of the data.

input  eclk ;           // serial clock
input  reset ;
input  transmit1 ;      // First transmit signal, to indicate to take
                        // data out of last FIFO stage
input  transmit2 ;      // Second transmit signal, to tell registers to
shift
input  [7:0] last ;     // Data from last stage of FIFO
output sout ;           // Serial out data

wire   inclk ;          // Combined clock (~eclk OR transmit1)
reg    [7:0] q ;        // Shift Register

assign inclk = ~eclk | transmit1;
                        // When transmit1 goes high and transmit2
                        // is low, this will cause the register to
                        // to take on the FIFO's data. Otherwise, with
                        // transmit2 high, the data will be shifted out
                        // serially.
always @(posedge inclk or posedge reset)
if (reset) q <= 8'b0;
else if (transmit2) begin
      q[0] <= 1'b0;
      q[1] <= q[0];
      q[2] <= q[1];
      q[3] <= q[2];
      q[4] <= q[3];
      q[5] <= q[4];
      q[6] <= q[5];
      q[7] <= q[6];
      end
else q <= last;

assign sout = q[7];

endmodule
```

Appendix C: Assembly Code

```
*Nick Bodnaruk and Andrew Ingram
*12-3-00
*This code interfaces the HC11 with the microps FPGA board via the SPI I/O
*subsystem of the HC11. It sends data to a FIFO on the FPGA, which then sends *the
data back to the HC11 for verification.

ddrd    EQU   $1009  *use EQU statements to make code more readable
spcr    EQU   $1028
spsr    EQU   $1029
spdr    EQU   $102A
parallel EQU  $1004


data1   EQU   $14
data2   EQU   $03
data3   EQU   $20
data4   EQU   $42

       org $d000

*rather than use FCB format, this makes clear what the data items are, and
*makes it easier to change them

       ldaa   #data1
       staa   $01
       ldaa   #data2
       staa   $02
       ldaa   #data3
       staa   $03
       ldaa   #data4
       staa   $04

       ldaa #%00111000      *configure port D data register
       staa ddrd
       ldaa #%01011100 *configure SPI control register
       staa spcr

       ldy #$0001
top:
       cpy #$0005    *when y= 5 all four data items have been sent
       beq sent      *if all data is sent, branch to recieve portion

       ldab #$01     *store 0001 in accumulator b
       stab parallel *write to the parallel port, tells FPGA that data is valid

       ldaa spsr     *read spsr to clear the flag
       ldaa 0,y
       staa spdr     *write to SPI data register
       staa 32,y     *store output to memory to check data

hold:  ldaa spsr     *wait until transfer is finished
       anda #$80
       beq hold

       iny           *increment y

       ldx #$0000    *give FPGA time to pass data into FIFO before next data

wait:  inx
       cpx #$0010    *compare x to 16, to give FPGA time
       bne wait


       bra top
```

```
sent:  ldy #$0001

get:   cpy #$0005    *when y= 5 all data is recovered
       beq done      *if all data is received, we are done

       ldab #$02     *set bit one to 1
       stab parallel   *write to parallel, tells FPGA to latch data fom the FIFO
       ldab #$04     *set bit two to 1
       stab parallel   *write to parallel

       ldaa #$77     *write bogus data to the serial port so we can read from FPGA
       staa spdr

hold2: ldaa spsr     *wait while transfer finishes
       anda #$80
       beq hold2

       ldaa spdr     *read data from the serial port
       staa 16,y     *store read data with an offset of 16 from y

       ldab #$00     *return parallel port to zeros
       stab parallel

       iny           *increment y
       bra get

done:  swi           * all data has been retrieved
```