EECS 150 - Components and Design Techniques for Digital Systems

Lec 19 – Putting it together

Case Study: A Serial Line Transmitter/Receiver

10-30-07

David E. Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

Outline

• Quick Review
• A Serial Line Transmitter/Receiver
  – Designing hardware to communicate over a single wire.
  – The data will be sent serially
  – The clocks on the two ends are completely asynchronous.
  – Communication protocol – RS232

Recall: Levels of Design Representation

Datapath + Control

• Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  – Inputs are Control Points
  – Outputs are signals
• Controller: State machine to orchestrate operation on the data path
  – Based on desired function and signals
Topics since MidTerm I

- Project
  - Clock domains, protocols, integration of digital subsystems
  - Controller design, interfaces, buffering, matching
- Timing
  - Propagation delays, setup, hold, critical path, fan-out, wires
- Static RAM
  - 6T cell, organization (decode, cells, row, col mux, drive & sense)
    - Shared data I/O, Read/Write Protocol,
- Dynamic DRAM
  - 1T cell, organization (decoders, row buffers, refresh)
    - Multiplexed address, RAS, CAS, protocols (page, static col, ...)
- Number Systems
  - Integer, Fix Point, Floating Point
  - Unsigned, 2s comp, sign mag., 1s comp, excess. Operations
- Arithmetic Circuits
  - Addition, Subtraction, Compare, Multiply
- Error detection, correction, coding
  - Parity, Hamming, SECDED, redundancy check, CRC

Motivation

- Data Transmission in asynchronous circuits
  - Handshaking
  - Oversampling
- Design techniques
  - Timing diagram
  - Top-down design
  - Bubble-and-arc
- Communication Protocol – RS232
  - One of the most heavily used standards ever developed for computing equipment.
  - Baud rate: 9600 ~ 56000 bps

Keypad & LCD screen

- Standard parts...

Problem Specification (1)

- Design two subsystems:
  - Transmitter
    - Takes input from a telephone-like keypad.
    - Sends a byte corresponding to the key over a single wire one bit at a time.
  - Receiver
    - Receives the serial data sent by the transmitter
    - Displays it on a small LCD screen
Problem Specification (2)

- RS232 protocol for formatting the data on the wire
- The wire is normally high – quiescent value
- A data begins with a start bit – low for one bit
- 8 data bits with msb first
- After the 8 bits, the wire must be high for at least one bit time before the next byte’s start bit – stop bit

Understanding the Specification (1)

- Two devices will be completely asynchronous to each other.
  - The single wire will carry only “data”
  - The receiving side should be able to determine when a start bit is starting.
  - If sampling the wire once every bit time we may just miss the start bit between two high values.
  - We need to have a faster clock that will sample the wire faster than once every bit cycle.
- Oversampling
  - sample multiple times during each bit time to make sure that we pick out the starting falling edge of each byte’s transmission.
  - *Oversampling* is a common technique in communication circuits

Understanding the Specification (2)

- LCD device is asynchronous.
  - There is no clock.
  - We have asynchronous control signals
- Robust and Modular
  - Need to make sure the data has been transmitted.
  - Data processing speeds are not the same.
- Handshaking
  - Need to make sure your data has been transferred to the destination.
  - 1. Sender: Request
  - 2. Receiver: Request Acknowledged
  - 3. Sender: OK...
  - 4. Receiver: Work done!!

Design Techniques

- Timing Diagram
  - Oversampling needs to pick up falling edge of input signal and interacts with counters.
  - Handshaking is between asynchronous circuits and synchronous circuits.
  - In general, timing diagram is very important in digital design.
- Top Down & Bubble-and-arc Diagram
  - Top Down design: define larger block and break it into smaller blocks.
  - Bubble-and-arc for finite state machines.
Implementation (1)

Sender
- Keyboard - input device
- KeyboardDecode – decodes the signals from the keypad and turns them into the appropriate character code for the button. Bridges KB and Sender domains.
- Sender – takes the byte and serially transmits it over the single wire

Receiver
- Display – LCD output
- DisplayControl – gives data and controls to the LCD appropriately to get the corresponding character to show up on the screen. Bridges domain.
- Receiver – observes the signal wire coming from the sender and determines when a byte has been received.

Implementation (2)

Keyboard (1)

Keyboard (2)
**KeyboardDecode**

- It decodes key presses into the 8-bit character code.
- Four-cycle handshake – robust and modular

**KeyboardDecode – Handshaking (1)**

- Handshake between the KeyboardDecode block and the Sender block.
- Send is raised first.
- Raise AckS in response.
- This in turn will be seen by the original block that is now assured its raising of the Send output has been observed. It then lowers Send.
- Lower AckS.

**KeyboardDecode – Handshaking (2)**

- Either block can take more time to do what it needs to do by delaying when it raises or lowers its signal.
- If data is being sent along with the handshake, the data should be held constant from when Send is raised to when the acknowledgement, via AckS, is received.

**KeyboardDecode – Handshaking (3)**

- What happens if a second key is pressed?
KeyboardDecode – Verilog

```verilog
module KeyboardDecode (ClkS, ResetS, R1, R2, R3, R4, C1, C2, C3, AckS, Send, DOut);

input ClkS, ResetS, AckS;
input R1, R2, R3, R4, C1, C2, C3;
output Send;
output [7:0] DOut;

reg [7:0] DOut;
reg Send;
wire KeyPressed;
assign KeyPressed = (R1 | R2 | R3 | R4) & (C1 | C2 | C3);

always @ (posedge ClkS) begin
    if (KeyPressed & !AckS) begin
        if (R1 & C1) DOut <= 8'h00110001; // code for 1
        else if (R1 & C2) DOut <= 8'h00110010; // code for 2
        else if (R1 & C3) DOut <= 8'h00110011; // code for 3
        else if (R2 & C1) DOut <= 8'h00110100; // code for 4
        else if (R2 & C2) DOut <= 8'h00110101; // code for 5
        else if (R2 & C3) DOut <= 8'h00110110; // code for 6
        else if (R3 & C1) DOut <= 8'h00110111; // code for 7
        else if (R3 & C2) DOut <= 8'h00111000; // code for 8
        else if (R3 & C3) DOut <= 8'h00111001; // code for 9
        else if (R4 & C1) DOut <= 8'h00101010; // code for *
        else if (R4 & C2) DOut <= 8'h00110000; // code for 0
        else if (R4 & C3) DOut <= 8'h00100011; // code for#
    end
end

We use ASCII codes because the display uses
```

KeyboardDecode – Verilog (Handshaking)

```verilog
always @ (posedge ClkS) begin
    if (ResetS)
        Send <= 0;
    else if (KeyPressed & !AckS)
        Send <= 1;
    else if (AckS)
        Send <= 0;
end
```

---

We use ASCII codes because the display uses
Sender

- It serializes data into the RS232 format.
- It implements the other half of the four-cycle handshake with the KeyboardDecode module.
- It sends 10 bits over the serial line for each key pressed on the keyboard.

Sender – Block Diagram & Bubble-and-arc

Sender – Verilog (1)

Module Sender (ClkS, ResetS, Send, DIn, AckS, TxD);
input ClkS, ResetS, Send;
input [7:0] DIn;
output AckS, TxD;
reg AckS;
reg [7:0] Data;
reg [3:0] BitCount;

always @ (posedge ClkS) begin
  if (ResetS) AckS <= 0;
  else if (~AckS & Send) AckS <= 1;
  else if (AckS & BitCount[3]) AckS <= 0;
end

Sender – Verilog (2)

always @ (posedge ClkS) begin
  if (Send & ~AckS) Data <= DIn;
  else if (AckS && BitCount != 0) Data <= {Data[6:0], 1'b0};
end

always @ (*) begin
  if (AckS & BitCount == 0) TxD = 0;
  else if (AckS) TxD = Data[7];
  else TxD = 1;
end
**Sender – Verilog (3)**

```verilog
always @(posedge ClkS) begin
    if (ResetS) BitCount <= 0;
    else if (Send & ~AckS) BitCount <= 0;
    else BitCount <= BitCount + 1;
end
```

**Announcements**

- Midterm on Thursday 11/1
  - 1-page nodes
- Midterm Review on Tuesday 10/30
  - 8:00-10:00 in 125 Cory
- Clarify HW due time

**Receiver – Verilog (1)**

```verilog
Module Receiver (ClockR, ResetR, RxD, AckR, DOut, Rcvd);
input ClockR, ResetR, RxD, AckR;
output [7:0] DOut;
output Rcvd;
reg [7:0] DOut, Data;
reg Rcvd, Receiving;
wire [3:0] BitCount;
wire [1:0] CycleCount;

always @(posedge ClockR) begin
    if (ResetR) Receiving <= 0;
    else if (~Receiving & ~RxD) Receiving <= 1;
    else if (Receiving && BitCount == 9) Receiving <= 0;
end
```

**Receiver**

- It needs to sample the input to determine when a start bit occurs
- Then, it stores the value of the 8 bits after the start bit
- After that, it passes them on to the DisplayControl module.
### Receiver – Verilog (2)

**Counter Cycle**

```
.Counter Cycle (.Clock(ClockR),
              .Reset(ResetR | (~Receiving & ~RxD)),
              .Set(1'b0),
              .Load(),
              .Enable(Receiving),
              .In(),
              .Count(CycleCount));
```

defparam Cycle.width = 2;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RxD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Receiver – Verilog (3)

**Counter Bit**

```
.Counter Bit (.Clock(ClockR),
              .Reset(ResetR | (~Receiving & ~RxD)),
              .Set(1'b0),
              .Load(),
              .Enable(Receiving & CycleCount == 3),
              .In(),
              .Count(BitCount));
```

defparam Bit.width = 4;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RxD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Receiver – Verilog (4)

**ShiftRegister SIPO**

```
.ShiftRegister SIPO (.PIn(),
                      .SIn(RxD),
                      .POut(Data), .SOut(),
                      .Load(),
                      .Enable(Receiving && CycleCount == 1), .Clock(ClkR),
                      .Reset(ResetR));
```

### Receiver – Verilog (5)

**Register DataRegister**

```
.Register DataRegister (.Clock(ClkR),
                         .Reset(ResetR),
                         .Set(),
                         .Enable(BitCount == 9 && CycleCount == 0),
                         .In(Data),
                         .Out(DOut))
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RxD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagrams

- **ClkS**
  - Start: X
  - D[7]: 0
  - D[6]: 1
  - D[0]: 2
  - Stop: 8

- **RxD**
  - Start: X
  - D[7]: 0
  - D[6]: 1
  - D[0]: 2

- **Receiving**
  - X

- **CycleCount**
  - X

- **BitCount**
  - X
  - 0
  - 1
  - 2
  - 8
  - 9

These diagrams illustrate the operation of the receiver components in Verilog, showing the states and transitions of ClkS, RxD, Receiving, CycleCount, and BitCount over time.
Receiver – Verilog (6)

```verilog
always @ (posedge ClkR) begin
    if (ResetR) Rcvd <= 0;
    else if (BitCount == 9 && CycleCount == 0) Rcvd <= 1;
    else if (AckR) Rcvd <= 0;
end
```

Display – LCD (1)

- Enable driven interface
- At a falling edge of the Enable signal, LCD interprets the mode and the data inputs.
- Setup Time and Hold Time of LCD < ~10ns
- Our Clock is much slower than this.

Display – LCD (2)

- The RS input is used to indicate how to interpret the data bits
- 0: Command  1: Character
- Whenever we reset our circuit, we need to execute the four operations below.

<table>
<thead>
<tr>
<th>Operation</th>
<th>RS</th>
<th>DB7...DB0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Display</td>
<td>0</td>
<td>0000 0001</td>
</tr>
<tr>
<td>Function Set</td>
<td>0</td>
<td>0011 0011</td>
</tr>
<tr>
<td>Display On</td>
<td>0</td>
<td>0000 1100</td>
</tr>
<tr>
<td>Entry Mode Set</td>
<td>0</td>
<td>0000 0110</td>
</tr>
<tr>
<td>Write Character</td>
<td>1</td>
<td>DDDD DDDD</td>
</tr>
</tbody>
</table>

DisplayControl

- Two main tasks:
  - Initialize LCD.
  - Display the characters received by the receiver module.
- Four-cycle handshake with the Receiver module using Rcvd and AckR
DisplayControl – FSM(Counter)

Module DisplayControl (ClkR, ResetR, Rcvd, CharRcvd, AckR, DB, RS, Enable);
input ClkR, ResetR, Rcvd;
input [7:0] CharRcvd;
output AckR, RS, Enable;
output [7:0] DB;

reg AckR;
reg initMode; //indicates if the initialize sequence is in progress
reg [7:0] DB;
reg[3:0] CS;

assign RS = ~initMode;

DisplayControl – Verilog (2)
always @ (posedge ClkR) begin // FSM/Counter
    if (ResetR) CS <= 0;
    else if (initMode) CS <= CS + 1;
end

always @ (posedge ClkR) begin // FSM
    if (ResetR) initMode <= 1; else if (initMode == 1 && CS == 9) initMode <= 0;
end

DisplayControl – Verilog (3)
always @ (posedge ClkR) begin
    if (initMode & CS == 0) DB <= 8'b00000001;
    else if (initMode & CS == 2) DB <= 8'b00110011;
    else if (initMode & CS == 4) DB <= 8'b00001100;
    else if (initMode & CS == 6) DB <= 8'b00000110;
    else if (~initMode & Rcvd) DB <= CharRcvd;
end
assign Enable = (initMode? CS[0] : AckR);

DisplayControl – FSM(Counter)

DisplayControl – Verilog (1)
### DisplayControl – Verilog (4)

```verilog
always @ (posedge ClkR) begin
    if (ResetR) AckR <= 0;
    else if (~initMode & Rcvd & ~AckR) AckR <= 1;
    else if (~initMode & ~Rcvd) AckR <= 0;
end
```

### Review: Design hierarchy

![Design hierarchy diagram]

### Summary

- **Point of this case study**
  - to show how to breakdown a larger problem into components
  - to show how to define the interfaces between components

- There are many ways to accomplish this, the design presented here is merely one of these.

- Most of the design choices were made so as to make the implementation easy to understand.