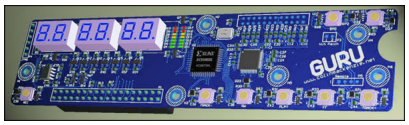
This is an example of a front panel interface for a digital audio recorder using a Xilinx XC9572XL CPLD interfaced with an Atmel AVR32AP7001 with the NGW100 Eval Kit. The CPLD provides "time multiplexed" sequencing for a 6 digit 7-segment LED display, 8 status LEDs, and detects input from 7 key switches generating a CPU interrupt on a key press or release. The CPLD interfaces with the AVR32AP using a simple 4-bit multiplexed bus. This provides a good example of designing a front panel interface that relieves the duty from the host CPU -- no polling!

To summarize, this controller handles:

- the display sequencing of 3x dual 7-segment LED displays (6 digits)
- the display of 8 regular status LEDs
- the input & detection of 7 switches (keys)
- bus transactions with the 32AP7001 processor over a simple 4bit (10 signal) bus



This PCB project as depicted also contains a Cirrus Logic CS4202 AC97 Codec sound chip for the AVR32AP that is not documented here.

Download Source Files

Source Files (tgz) (56.67 kB) Source Files (zip) (54.44 kB)

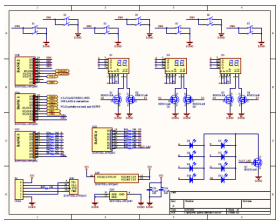
This file includes the Verilog source code, testbench, and output files for the controller. This source code was developed using Icarus verilog in Linux. To compile and run the testbench simply unpack the archive to an empty folder and run make. (Ensure you have the Icarus verilog compiler package installed.) The included Makefile runs the Icarus compiler and processor with the proper arguments. You should see the output from the testbench results. There is also a vcd dumpfile generated for gtkwave so you can look at the waveform timing diagrams.

Front Panel Schematics

The controller is based on the following schematics.These schematics are part of an expansion board I built for the NGW100 evaluation kit. This expansion board provides an AC97 sound codec to the AVR32AP processor. The full schematics and PCB artwork for this expansion

board will be posted at a later date. The expansion board with the ac97 codec and this front panel interface will also be available for purchase along with, (in accordance with GNU licensing,) freely available open-source code to implement a digital voice recorder using the SD card and the speex, mp3 or aac codec.

Download ac97 Front Panel Schematics as PDF



The Display Sequencer

To provide outputs from the CPLD for each of the 7 segments of each of the digits would require more pins than available on the device (48 to be exact), and would also be very ineffecient. The standard practice is to route the 7 segment pins (denoted as A-G,DP=dot) to each digit directly (daisy chain), then control the state of each digit's common cathode or ground pin. Thus even though each digit recieves the same signal, only the digit with the cathode pin active becomes visible, the cathodes on all other digits are floating or hi-impedance. (Note: some digit displays are common anode.)

To give the user the effect that all digits are being displayed at once we sequence the A-G signals and the cathode control signal very quickly, much faster than the speed of the human eye. Transistors are used between the digit's common cathode and ground to activate that digit. In my schematics I use mosfet's for low on-resistance and fast switching.

In this controller, the digit memory is 4-bits/digit. The CPU writes the value of the display via the 4-bit bus. The display sequencer has a 3-bit (0-7) counter which addresses the digit memory and outputs the stored value through a "hex to 7-segment display" decoder. This decoder takes the 4-bit input word (as hex) and converts it to the A-G signals for the 7-segment display.

Now, since we also have 8 status LEDs, why not control them with the display sequencer as well! 8 LED's cost us only 1 extra enable pin and a transistor, saving us 7 pins if we had to use 1 pin per LED. This makes our display sequencer a little more complex however, since the A-F outputs are driven by the 7-segment decoder based on only a 4-bit input. We dont want to display digit-like output on our status LEDs so the sequencer must bypass the 7-segment decoder and output 8bits directly when sequencing the status LEDs. This also means concatenating the two memory locations too, for an 8-bit value. Overall, not a major problem for our verilog program or CPLD.

The displace sequencer contains the following signals:

DISP_CLK - Input, the display sequencer clock, usually devised from system CLK, display sequences each DISP_CLK cycle.

DISPLAY[7:0] - Output, the decoded A-G+DP output to the 7-segment display.

DISP_SELECT[7:0] - Output, the select lines for the common cathode pins of digits 1-6, plus the LED status cathode enable.

The input keys

The input task is quite simple. When the CPU reads a specific memory address (the BUTTONS1 or BUTTONS2 location), we won't fetch the value from the memory but rather directly from the
switch inputs. Also, so as not to require polling by the CPU, if any switch is pressed or released we set the INTR output flag. This alerts the processor to check the BUTTONSx registers. The INTR flag is automatically reset when the CPU reads the last BUTTONS location.

If we were to have a large amount of keys we would use a switch matrix like in real keyboards and use less input pins, but for only 7 switches we can't save much.

keys[6:0] are the 7 key inputs from the signals.

The 4-bit bus

The CPU communicates with this controller over a simple 4-bit bus. This bus uses a multiplexed address/data bus. The desired read/write address is first latched into the controller

using lda pin (LoaD Address). Afterwhich, one or more RD/WR strobes are used to transfer data in and out of the device. After each RD or WR cycle, the device's internal address pointer is incremented. Thus, you don't have to reload the address if you are reading or writing from/to consecutive range of memory.

We could have used a simple 2 wire bus protocol like i2c if we were really short on cpu and/or device pins. However, this would take more latches to implement, and latches are in short supply in the Xilinx 9572 parts. The 10-pin interface is pretty simple and efficient. If you have the time, another example of a common 4-bit bus is the LPC bus as used in modern motherboards and the older XBox and is a good example of an efficient 4bit multiplexed bus! The LPC bus reduces the old ISA bus to about 10 pins without sacrifices in speed or features.

The 4-bit bus has the following signals:

CLK - Input, The bus clock

RST - Input, Resets the device

DA[3:0] - Bidirectional, The 4-bit multiplexed address/data bus

RD - Input, Active low during a read cycle

WR - Input, Active low during a write cycle

LDA - Input, Active low to latch DA[3:0] into the device's internal address latch register

INTR - Output, goes low when a key is pressed or released until the CPU reads the BUTTONS1 register

Memory Map

The following map outlines the registers internal to the device and thier read/write access.

00 R/W: DIGIT 0
01 R/W: DIGIT 1
02 R/W: DIGIT 2
03 R/W: DIGIT 3
04 R/W: DIGIT 4
05 R/W: DIGIT 5
06 R/W: Status LEDs 4-7
07 R/W: Status LEDs 3-0
08 R/W: DP2 5-4 7-Segment Display Dots
09 R/W: DP1 3-0 7-Segment Display Dots
0a R: BUTTONS 6-4
0b R: BUTTONS 3-0

Timing

Each step in the following timing diagrams is a cycle of the CLK. You can safely have many clock cycles occur for each step (relaxed timing requirements), but you must have at least one.

Writing to memory

1: Set da to the 4-bit memory address, set lda low
2: Set lda high, set da to data value, set wr low
3: Set wr high
[if we wish to continue writing in the next memory location, we repeat]
4: Set da to data value, set wr low
5: set wr high
[repeat from step 4 for more data]

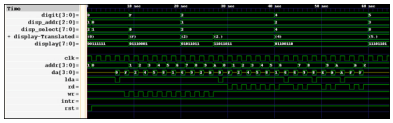
Reading from memory

1: Set da to the 4-bit memory address, set lda low
2: Set lda high, set rd low
3: Read data from da, set rd high
[if we wish to continue reading from the next memory location, we repeat]
4: Set rd low
5: Read data from da, set rd high
[repeat from step 4 for more data]

Timing Diagram

See the Source Files Archive for larger timing diagram and a GTKWave save file and VCD file for more in depth look at the timing sequences.

In this diagram the top 5 rows represent the Display Sequencer output. "display-translated" is the same signals as the "display[7:0]" signals, only it is translated by GTKWave using a *translation filter file*
. This translation filter file translates the 7-segment output back into hex value including the DP (decimal point).



Using Icarus Verilog

I would like to take a moment to speak about Icarus verilog, the free open-source verilog compiler. At first glance I had some problems using Icarus, mostly due to not having a convenient GUI to use with debug and waveform windows. I've used ModelSim previously but this package is expensive and doesn't provide an easy way for beginners to download and try out my code. My goals in posting my verilog/HDL projects here is to get more people interested and working with Hardware Descriptive Languages as I believe certain problems are best solved in programmable logic. Furthermore, programming programmable logic devices are so much easier now-a-days than ever before, no good systems engineer should feel shy to use them. For this reason, I resolved to using Icarus Verilog for every project I will post online.

Now that I have become more fluent in writing my test benches and using the simulation/debug functions provided by verilog, coupled with a Makefile, I find writing modules with Icarus very quick and easy now, and just as quick as with ModelSim. So have no fear, just like Icarus, fly close to the sun...and soon you'll make it over the Icarus humps.

Icarus also isn't fully Verilog2001 compliant. It is missing a few features like the ***generate*** statement and bit-level accessing of word memory arrays. Both of these features could simply a bit of the following code, but in synthesis it probably wouldn't make much of a difference in latch/routing resources.

Verilog Source Code

```
 module ac97ui(
  clk, rst, da, rd, wr, lda, intr,  // 4-bit bus interface
  disp_clk, display, disp_select,  // display ports
  keys
  );

input clk, rst, rd, wr, lda;    // our system clock
output intr;    // interrupt output when a key is pressed or released
inout [3:0] da; // bidirectional address/data bus

input disp_clk; // seperate display clock
output [7:0] display;   // the decoded 7-segment display output
output [7:0] disp_select;   // disp_select decoder decodes only when data is not 0xf

// the keys/buttons, our input switches
input [6:0] keys;

// the names of each of our address locations
parameter D0      = 4'b0000;  // Digit 0
parameter D1      = 4'b0001;
parameter D2      = 4'b0010;
parameter D3      = 4'b0011;
parameter D4      = 4'b0100;
parameter D5      = 4'b0101;
parameter LEDS2   = 4'b0110;  // Status LEDs [7:4]
parameter LEDS1   = 4'b0111;  // Status LEDs [3:0]
parameter DP2     = 4'b1000;  // Digit Dots [5:4] (bits 6,7 are dont matter)
parameter DP1     = 4'b1001;  // Digit Dots [3:0]
parameter BUTTONS2 = 4'b1010; // Buttons [6:4]    (bit 7 always reads as 1)
parameter BUTTONS1 = 4'b1011; // Buttons [3:0]


// our memory core consisting of Instruction Memory, Register File and an ALU working (W)
register
reg [ 3:0 ] addr;          // our address register
reg [ 3:0 ] mem[7:0] ;    // our digit memory
reg [ 5:0 ] dpmem;      // memory for our display dots (must be seperate memory because of
icarus verilog bit-select limitation)

reg intr = 1;   // active low when a key is pressed

reg [ 2:0 ] disp_addr;
wire [ 3:0 ] digit; // the value of the addressed hex digit
wire [ 6:0 ] hexdecoder_out;
wire dp_out;  // our dot value for the currently sequenced digit
```

```
// instantiate our hex to 7 segment decoder
hex_to_7segment hexdecoder( digit, hexdecoder_out );

// this task resets our device
task dev_reset;
begin
  {mem[0],mem[1],mem[2],mem[3],mem[4],mem[5],mem[6],mem[7]} = 32'b0;
  dpmem = 6'b000000;
  intr = 1;
  addr = 4'b0000;
  disp_addr = 4'h00;
end
endtask

// the initial state, not synthesized, so a reset cycle is recommended/required
initial #1 begin  // reset on second timestep
  dev_reset;
end

/* 4-bit Simple Bus
 */

// during a read, we get the value from memory or directly from the switches
assign da =
    (rd)    ? 8'bz :
    (addr==DP2) ? {2'b11, dpmem[5:4]} :
    (addr==DP1) ? dpmem[3:0] :
    (addr==BUTTONS2) ? { 1'b1, keys[6:4] } :
    (addr==BUTTONS1) ? keys[3:0] :
    mem[ addr ];

// at each clock cycle we manage the state of the 4bit bus
always @ (clk, rst)
begin
if(!rst)
  dev_reset;  // keep the device in the reset state
else
  begin
  if(!lda)
    addr
```