XUPv5 LCD

The LCD module is a TM162VDCW6, but the datasheet is useless. It's controlled by a ks0066u chipset, the datasheet for which is much more useful (http://www.lcd-module.de/eng/pdf/zubehoer/ks0066.pdf). This shows that the module has 8 data lines and 3 control lines, but is capable of operating in 4-bit mode with only half of the data lines connected, and it is used in this mode on the XUPv5 board. It's connected via a CPLD that's used simply for level shifting. The appropriate pins are mapped to nets in the XUPv5 UCF file:

```
NET LCD_FPGA_DB4 LOC="T9"; # Bank 12, Vcco=3.3V, DCI using 49.9 ohm resistors
NET LCD_FPGA_DB5 LOC="G7"; # Bank 12, Vcco=3.3V, DCI using 49.9 ohm resistors
NET LCD_FPGA_DB6 LOC="G6"; # Bank 12, Vcco=3.3V, DCI using 49.9 ohm resistors
NET LCD_FPGA_DB7 LOC="T11"; # Bank 12, Vcco=3.3V, DCI using 49.9 ohm
resistors
NET LCD_FPGA_E LOC="AC9"; # Bank 22, Vcco=3.3V, DCI using 49.9 ohm resistors
NET LCD_FPGA_RS LOC="J17"; # Bank 3, Vcco=2.5V, No DCI
NET LCD_FPGA_RW LOC="AC10"; # Bank 22, Vcco=3.3V, DCI using 49.9 ohm
resistors
```

The three control lines are RS (register select), RW (read/write) and E (read/write enable). From the datasheet:

```
This chip has both kinds of interface type with MPU: 4-bit bus and 8-bit
bus.  4-bit bus and 8-bit bus are selected by the DL bit in the instruction
register.
During read or write operation, two 8-bit registers are used.  One is the
data register (DR), and the other is the instruction register (IR).  The data
register (DR) is used as a temporary data storage place for being written
into or read from DDRAM/CGRAM.  The target RAM is selected by RAM address
setting instruction.  Each internal operation, reading from or writing into
RAM, is done automatically.  Thus, after MPU reads DR data, the data in the
next DDRAM/CGRAM address is transferred into DR automatically. Also, after
MPU writes data to DR, the data in DR is transferred into DDRAM/CGRAM
automatically.
The Instruction register(IR) is used only to store instruction codes
transferred from MPU.  MPU cannot use it to read instruction data.
```

The following table, from the datasheet, shows how the RS and RW lines are used:

RS	RW	Operation
L	L	Instruction Write operation (MPU writes Instruction code into IR)
L	H	Read Busy flag(DB7) and address counter (DB0 to DB6)
H	L	Data Write operation (MPU writes data into DR)
H	H	Data Read operation (MPU reads data from DR)

More from the datasheet:

Busy Flag (BF)

BF = "High", indicates that the internal operation is being processed. So during this time the next instruction cannot be accepted. BF can be read through DB7 port when RS = "Low" and R/W = "High" (Read Instruction Operation). Before executing the next instruction, be sure that BF is not "High".

Address Counter (AC)

The address Counter (AC) stores DDRAM/CGRAM addresses, transferred from IR. After writing into (reading from) DDRAM/CGRAM, AC is automatically increased (decreased) by 1. When RS = "Low" and R/W = "High", AC can be read through ports DB0 to DB6.

Display Data RAM (DDRAM)

DDRAM stores display data of maximum 80×8 bits (80 characters). DDRAM address is set in the address counter (AC) as a hexadecimal number (AC6 downto AC0).

Note that on the XUPv5 board, the Busy Flag is completely useless because the presence of the CPLD effectively makes the LCD module write-only. For that reason, the RW line can be tied low. I wonder why they bothered routing it?!

On this 2-line display, the top line is at addresses 00-0F and the bottom line is at 40-4F, though the line buffers actually run from 00-27 and 40-67 (40 characters per line), and can be scrolled.
Character generation is from ROM (CGROM) or RAM (CGRAM), giving the opportunity for user-defined character sets.

Commands:

Command	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Comments	Execution Time
Clear Display	0	0	0	0	0	0	0	0	0	1	Also resets cursor to top left, and AC to 00	1.53ms
Return Home	0	0	0	0	0	0	0	0	0	x	Resets cursor to top left, and AC to 00	1.53ms
Entry Mode	0	0	0	0	0	0	0	1	ID	SH	Sets the operations to be performed after each entry (write to DDRAM) SH=0: Moves cursor right/left and inc/ dec AC (ID=1/0) SH=1: Shifts entire display left/right (ID=1/0)	39µs
Display Control	0	0	0	0	0	0	1	D	C	B	D=1/0: Display on/off C=1/0: Cursor on/off B=1/0: Cursor blink on/off	39µs

Cursor or Display Shift	0	0	0	0	0	1	SC	RL	x	x	Moves the cursor or scrolls the display, immediately SC=0: Move cursor right/left and inc/dec AC (RL=1/0) SC=1: Shift display right/left (RL=1/0), cursor moves with display	39µs
Function Set	0	0	0	0	1	DL	N	F	x	x	DL=0/1: Select 4/8 bit mode N=0/1: Select 1/2 line mode F=0/1: Select 5x8/5x11 dot characters	39µs
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address to AC, allowing access to CGRAM	39µs
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address to AC, allowing access to DDRAM	39µs
Read Busy Flag & Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	BF=0/1: Device free/busy AC: Current address	0µs
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write to current location (see 'Set CGRAM/DDRAM Address') AC is then modified according to Entry Mode	43µs
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read from current location (see 'Set CGRAM/DDRAM Address') AC is then modified according to Entry Mode - some caveats, see datasheet	43µs

Controlling the Screen: Software Solution

The example on http://www.fpgadeveloper.com/2008/10/microblaze-16x2-lcd-driver.html shows how a GPIO peripheral can be used to hook a MicroBlaze directly to the LCD module and get it to do stuff. It's a good example in that it's simple and it works, unlike most of the Xilinx crap. But it might actually serve pretty well as an example of where *not* to put the boundary between hardware and software: it spends most of its time in software-controlled wait loops. Clearly that's poor use of the MicroBlaze's capabilities, and devolving

the interfacing to a hardware module is the way around that.

Controlling the Screen: Hardware Solution

The first hardware test was standalone (using the ISE) before integrating it as a peripheral into an EDK project.

The Hardware

This was implemented hierarchically, with an 8-bit driver feeding a 4-bit driver.

4-bit Driver

It's pretty simple to write a driver that will present 4-bit data (plus RS and RW) to the LCD, controlling the 'E' (data strobe) line and producing a busy signal for the duration. Here's my attempt:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Driver_4Bit is
    Generic (
                CLK_MHZ : NATURAL := 33
            );
    Port (
                clk : in STD_LOGIC;
                RS_in : in  STD_LOGIC;
                data_in : in  STD_LOGIC_VECTOR (3 downto 0);
                load : in  STD_LOGIC;
                ack_busy : out STD_LOGIC;
                LCD_data : out  STD_LOGIC_VECTOR (3 downto 0);
                LCD_RS : out  STD_LOGIC;
                LCD_RW : out  STD_LOGIC;
                LCD_E : out  STD_LOGIC
            );
end Driver_4Bit;

architecture Behavioral of Driver_4Bit is

    type STATE_TYPE is ( IDLE, SETUP, ACTIVE, HOLD,
WAIT_FOR_LOAD_LOW );
    subtype STATE_TIME_TYPE is INTEGER range 0 to CLK_MHZ; -- max
state time 1us
    type STATE_TIMINGS_TYPE is array(STATE_TYPE) of
STATE_TIME_TYPE;
```

```
    constant state_timings : STATE_TIMINGS_TYPE := (
        IDLE => 0,
        SETUP => STATE_TIME_TYPE(CLK_MHZ * 60 / 1000),    -- 60ns
min time in the SETUP state
        ACTIVE => STATE_TIME_TYPE(CLK_MHZ * 450 / 1000),  -- 450ns
min time in the ACTIVE state
        HOLD => STATE_TIME_TYPE(CLK_MHZ * 450 / 1000),    -- 450ns
min time in the HOLD state
        WAIT_FOR_LOAD_LOW => 0
    );
    signal state_time, state_time_load_value : STATE_TIME_TYPE :=
0;
    signal state_time_load : STD_LOGIC := '0';
    signal state_locked : STD_LOGIC;
    signal state : STATE_TYPE := IDLE;
    signal data_reg : STD_LOGIC_VECTOR (3 downto 0) :=
(others=>'0');
    signal RS_reg : STD_LOGIC := '0';

begin

with state_time select
    state_locked    <=    state_time_load when 0,
                            '1' when others;

-- Lock state transitions while states are active
TICK_PROC: process (clk) is
begin
    if rising_edge(clk) then
        if state_time_load = '1' then
            state_time <= state_time_load_value;
        elsif state_time > 0 then
            state_time <= state_time - 1;
        end if;
    end if;
end process TICK_PROC;

-- State machine
STATE_PROC: process (clk, state_time, load, state) is
    variable next_state : STATE_TYPE;
begin
    if rising_edge(clk) then
        state_time_load <= '0';
        if state_locked = '0' then
            next_state := state;
            case state is
                when IDLE =>
                    if load = '1' then
                        data_reg <= data_in;
                        RS_reg <= RS_in;
                        next_state := SETUP;
                    else
```

```
                        next_state := IDLE;
                    end if;
                when SETUP =>
                    next_state := ACTIVE;
                when ACTIVE =>
                    next_state := HOLD;
                when HOLD | WAIT_FOR_LOAD_LOW =>
                    if load = '1' then
                        next_state := WAIT_FOR_LOAD_LOW;
                    else
                        next_state := IDLE;
                    end if;
                end case;
            end if;
            state <= next_state;
            state_time_load_value <= state_timings(next_state);
            if state /= next_state then
                state_time_load <= '1';
            end if;
        end if;
    end if;
end process STATE_PROC;

with state select
    LCD_E <=    '1' when ACTIVE,
                '0' when others;

with state select
    ack_busy <=    '0' when IDLE,
                    '1' when others;

LCD_RS <= RS_reg;
LCD_RW <= '0';
LCD_data <= data_reg;

end Behavioral;
```

It lacks comments (oops) but is fairly simple. One process, named STATE_PROC, handles the state transitions for a 5-state Moore machine. Transition from 'IDLE' to 'SETUP' occurs when 'load' is pulled high; 'SETUP' leads through 'ACTIVE' to 'HOLD' and finally to 'WAIT_FOR_LOAD_LOW', where the transition back to 'IDLE' occurs only when 'load' becomes low again. The 'ack_busy' output is held high whenever the state is not IDLE. Data is loaded onto the LCD lines in the 'SETUP' state, then the 'E' line on the LCD module is pulled high in the 'ACTIVE' state and low again for the 'HOLD' state. State timings are achieved by using a signal which, when asserted, prevents the state transitions from occurring; the TICK_PROC process and the selected assignment just above it are responsible for generating this signal. The state timer is reloaded automatically on state transitions by the assertion of the 'state_time_load' signal, which is generated by the STATE_PROC process on every transition.

8-bit Driver

Wrapping the 4-bit driver to create an 8-bit driver should also be simple - but is rendered complicated by one factor. When the LCD screen is initialised, it needs to be sent some 4-bit packets (see the datasheet) to put it into 4-bit mode. Since the 8-bit driver hides the 4-bit functionality from anything that wraps it, it must generate all of the required setup signals and timings itself. Here is my code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Driver_8Bit is
    Generic (
                CLK_MHZ : NATURAL := 33
            );
    Port (
                clk : in STD_LOGIC;
                data_in : in  STD_LOGIC_VECTOR (7 downto 0);
                RS_in : in  STD_LOGIC;
                load : in STD_LOGIC;
                ack_busy : out  STD_LOGIC;
                LCD_data : out  STD_LOGIC_VECTOR (3 downto 0);
                LCD_RS : out  STD_LOGIC;
                LCD_RW : out  STD_LOGIC;
                LCD_E : out  STD_LOGIC
            );
end Driver_8Bit;

architecture Behavioral of Driver_8Bit is
    type STATE_TYPE is ( INIT, INIT_WAIT_1, INIT1, INIT1_WAIT,
INIT2, INIT2_WAIT, INIT3, INIT_WAIT_2,
                                IDLE, MSB, MSB_WAIT, LSB, LSB_WAIT,
WAIT_FOR_LOAD_LOW );
    signal state : STATE_TYPE := INIT;
    subtype STATE_TIMER_TYPE is INTEGER range 0 to
(CLK_MHZ*50*1000); -- 50ms max wait
    shared variable state_timer : STATE_TIMER_TYPE := 0;
    signal driver_busy : STD_LOGIC := '0';
    signal driver_load : STD_LOGIC := '0';
    signal data_reg : STD_LOGIC_VECTOR (7 downto 0) :=
(others=>'0');
    signal data_nibble : STD_LOGIC_VECTOR (3 downto 0) :=
(others=>'0');
    signal RS_reg : STD_LOGIC := '0';
begin

-- State machine
```

```
STATE_PROC: process (clk, load, state) is
begin
    if rising_edge(clk) then
        if state_timer /= 0 then
            state_timer := state_timer - 1;
        end if;
        case state is
            when INIT =>
                if state_timer = 0 then
                    state_timer := CLK_MHZ * 50 * 1000; -- 50ms
initial delay
                    state <= INIT_WAIT_1;
                end if;
            when INIT_WAIT_1 =>
                if state_timer = 0 then
                    state <= INIT1;
                end if;
            when INIT1 =>
                if driver_busy = '1' then
                    state <= INIT1_WAIT;
                end if;
            when INIT1_WAIT =>
                if driver_busy = '0' then
                    state <= INIT2;
                end if;
            when INIT2 =>
                if driver_busy = '1' then
                    state <= INIT2_WAIT;
                end if;
            when INIT2_WAIT =>
                if driver_busy = '0' then
                    state <= INIT3;
                end if;
            when INIT3 =>
                if driver_busy = '1' then
                    state_timer := CLK_MHZ * 15 * 1000; -- 15ms
delay after initialisation
                    state <= INIT_WAIT_2;
                end if;
            when INIT_WAIT_2 =>
                if state_timer = 0 then
                    if load = '1' then
                        state <= WAIT_FOR_LOAD_LOW;
                    else
                        state <= IDLE;
                    end if;
                end if;
```

```
            when IDLE =>
                if driver_busy = '0' and load = '1' then
                    data_reg <= data_in;
                    RS_reg <= RS_in;
                    state <= MSB;
                end if;
            when MSB =>
                if driver_busy = '1' then
                    state <= MSB_WAIT;
                end if;
            when MSB_WAIT =>
                if driver_busy = '0' then
                    state <= LSB;
                end if;
            when LSB =>
                if driver_busy = '1' then
                    state <= LSB_WAIT;
                end if;
            when LSB_WAIT =>
                if driver_busy = '0' then
                    if RS_reg = '1' then
                        state_timer := CLK_MHZ * 45; -- Write
operations take 43us
                    elsif data_reg (7 downto 2) = "000000" then
                        state_timer := CLK_MHZ * 2000; -- Clear or
Home operations take 1.53ms in theory
                    else
                        state_timer := CLK_MHZ * 40; -- Config
operations take 39us
                    end if;
                    if load = '1' or state_timer /= 0 then
                        state <= WAIT_FOR_LOAD_LOW;
                    else
                        state <= IDLE;
                    end if;
                end if;
            when WAIT_FOR_LOAD_LOW =>
                if load = '1' or state_timer /= 0 then
                    state <= WAIT_FOR_LOAD_LOW;
                else
                    state <= IDLE;
                end if;
        end case;
    end if;
end process STATE_PROC;

with state select
```

```
    ack_busy <=    '0' when IDLE,
                    '1' when others;

with state select
    driver_load <=    '1' when MSB | LSB | INIT1 | INIT2 | INIT3,
                        '0' when others;

with state select
    data_nibble <=    data_reg (7 downto 4) when MSB | MSB_WAIT,
                        data_reg (3 downto 0) when LSB | LSB_WAIT,
                        "0010" when INIT1 | INIT1_WAIT | INIT2 |
INIT2_WAIT | INIT3 | INIT_WAIT_2,
                        (others=>'0') when others;

driver: entity work.Driver_4Bit(Behavioral)
    generic map (
        CLK_MHZ => CLK_MHZ
    )
    port map (
        clk => clk,
        data_in => data_nibble,
        load => driver_load,
        RS_in => RS_reg,
        ack_busy => driver_busy,
        LCD_data => LCD_data,
        LCD_RS => LCD_RS,
        LCD_RW => LCD_RW,
        LCD_E => LCD_E
    );

end Behavioral;
```

The state machine in this module (controlled by the STATE_PROC process) has 14 states, but only 6 of them are required once the module is initialised - the rest are for the setup. It's pretty simple though. Note that the implementation could be made more efficient by splitting the state timer (implemented as a variable within STATE_PROC) off into a separate process, like in the 4-bit driver, which would then stand more chance of inferring a loadable down-counter rather than a subtractor and a register. But I couldn't be bothered.

Feeder (for testing in ISE)

The 'Feeder' module is effectively a synthesisable testbench for the 8-bit (and hence 4-bit) drivers. Of course the drivers were also testbenched independently using the ISE simulator, but this allows the drivers above to be implemented on an FPGA and tested for real:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Feeder is
    Port (
        CLK_33MHZ_FPGA : in STD_LOGIC;
        LCD_lines : out STD_LOGIC_VECTOR (6 downto 0)
    );
end Feeder;

architecture Behavioral of Feeder is
    type STACK_TYPE is array (0 to 10) of STD_LOGIC_VECTOR(8 downto
0);
    constant stack : STACK_TYPE :=
        (
            "000101100",-- setup
            "000001111",-- display on, blinking cursor
            "000000001",-- clear display
            "000000110",-- entry mode set
            "101001000",-- H
            "101100101",-- e
            "101101100",-- l
            "101101100",-- l
            "101101111",-- o
            "100100001", -- !
            "100000000" -- nop
        );
    signal data : STD_LOGIC_VECTOR(8 downto 0) := (others=>'0');
    signal load : STD_LOGIC := '0';
    signal ack_busy : STD_LOGIC;
    signal clk : STD_LOGIC;

--    signal control0 : STD_LOGIC_VECTOR(35 DOWNTO 0);
    signal LCD_lines_buffer : STD_LOGIC_VECTOR (6 downto 0);
    signal info_buffer : STD_LOGIC_VECTOR (1 downto 0);

--    component ila
--    PORT (
--        CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
--        CLK : IN STD_LOGIC;
--        TRIG0 : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
--        TRIG1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0));
--    end component;

--    component chipscope
--    PORT (
--        CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0));
```

```
--    end component;

begin

clk <= CLK_33MHZ_FPGA;

process (clk) is
    variable stage : natural := 0;
    variable from : natural := 0;
begin
    if rising_edge(clk) then
        if from < 10 then
            case stage is
                when 0 =>
                    if ack_busy = '0' then
                        stage := 1;
                    end if;
                when 1 =>
                    data <= stack(from);
                    load <= '1';
                    stage := 2;
                when 2 =>
                    if ack_busy = '1' then
                        stage := 3;
                    end if;
                when others =>
                    load <= '0';
                    if ack_busy = '0' then
                        stage := 1;
                        from := from + 1;
                    end if;
            end case;
        end if;
    end if;
end process;

LCD_lines <= LCD_lines_buffer;
info_buffer(1) <= ack_busy;
info_buffer(0) <= load;

driver: entity work.Driver_8Bit(Behavioral)
    generic map (
        CLK_MHZ => 33
    )
    port map (
        clk => clk,
        data_in => data(7 downto 0),
```

```
        load => load,
        RS_in => data(8),
        ack_busy => ack_busy,
        LCD_data => LCD_lines_buffer(3 downto 0),
        LCD_RW => LCD_lines_buffer(4),
        LCD_RS => LCD_lines_buffer(5),
        LCD_E => LCD_lines_buffer(6)
    );

--ila_instance : ila
--  port map (
--    CONTROL => CONTROL0,
--    CLK => CLK,
--    TRIG0 => LCD_lines_buffer,
--    TRIG1 => info_buffer);

--chipscope_instance : chipscope
--  port map (
--    CONTROL0 => CONTROL0);

end Behavioral;
```

The commented-out lines are for instantiating a chipscope core and an integrated logic analyser, for testing. Once again this is a very simple state machine; so simple, in fact, that I couldn't be bothered to create a state variable type so it just uses a natural (non-negative integer) number. The state machine waits for the 'ack_busy' line on the driver to be low, then asserts the next data word and the 'load' line, and finally de-asserts the 'load' line when 'ack_busy' goes high. This is not the neatest VHDL in the world, but it only exists as a test.

Creating a Peripheral

This sucks quite a lot in EDK 10.1. Any instructions you find for 8.1 simply won't work anymore.

The Process

If the 'create or import peripheral' wizard is run, in 'create' mode, with the defaults accepted, it'll make a peripheral with two VHDL sources. If the peripheral is called 'test', there'll be 'test.vhdl' and 'user_logic.vhdl'. The former is a wrapper for the latter, and it also serves to wrap other bits of the PLB so the user_logic file doesn't need to be aware of them.
The VHDL is pretty self-explanatory. To add a data port to the peripheral (eg. for external connectivity), add it to the user_logic block and then to the wrapper, and add it to the port map in the wrapper too. However, XPS will NOT notice that you've done that, and there's no way to tell it - the peripheral must then be imported (using the wizard) *on top of itself*.

Start the wizard in 'import' mode, bash in the name of the peripheral, and tick 'use version', ensuring that the version number matches that which you gave the peripheral originally. You'll get a warning about overwriting the peripheral's files, which you can accept. Then tick 'HDL source files', and select 'Use existing Peripheral Analysis Order file (*.pao)", browsing to the pao file in the 'data' folder inside the peripheral. After the HDL summary screen, the wizard will parse the HDL and report any errors it finds. Next choose 'PLBv46 Slave (PLB)' as the bus interface, and click through to the end, disabling interrupt support on the way. If other VHDL sources are to be integrated into the peripheral, they must be included in the PAO file before re-importing the peripheral. The lines in the file are strictly in reverse hierarchical order, so the additional filenames will be placed just above user_logic.

The peripheral can now be added from the 'IP catalog'. Expanding it in the 'bus interfaces' view, it can be connected to mb_plb; in the 'addresses' view, give it 64k and press 'generate addresses'. Now under the 'ports' view, expand the peripheral and make connections to the other (non-PLB) ports. If off-device connections are required, *name the net first by typing in the drop-down box* (the auto-generated names are needlessly huge) and then select 'make external'. The port will appear in the 'external ports' list, a net will be generated and assigned to link the internal port to the external port, and an external port name will be generated (named the same as the net but with a '_pin' suffix).

The UCF needs updating to include pin maps, so in the 'project' view, find the UCF and right-click to edit. The net names are those given to the external port (ending '_pin') in the previous stage.

An Example

This example system uses a Microblaze core at 125MHz, with only the 'rs232_uart_1' peripheral enabled in the Base System Builder, and every other peripheral disabled. The automatic 'memory self-test' and 'peripheral self-test' options are also disabled, though this isn't vital. Create a peripheral called LCD using the steps above. However, the peripheral needs to carry a FIFO so that the C code will never be kept waiting. So choose 'add to an XPS project', name the peripheral 'lcd', version 1.00a, and choose the PLB for connection. Next deselect every option except for 'Read/Write FIFO'. Note that the usual way of interacting with a peripheral is to use 'user logic software registers', but if you have a FIFO there's no need. Leave 'burst and cache-line support' switched off, and on the next screen deselect the Read FIFO entirely (the peripheral is write-only) and deselect 'Use Packet Mode' on the Write FIFO. Note that you must leave vacancy calculation enabled - I couldn't make the FIFO work at all without it! The defaults are fine for the remainder of the wizard.

Now, external ports are required for the LCD itself - and these need adding to the wrapper ([project]\pcores\lcd_v1_00_a\hdl\vhdl\lcd.vhd) and to user_logic.vhd in the same location, and then wiring through between the two. Rather than paste the entire code here, this is what needs to be added:

lcd.vhd: in entity declaration

```
    -- ADD USER PORTS BELOW THIS LINE ------------------
    LCD_data : out  STD_LOGIC_VECTOR (3 downto 0);
```

```
    LCD_RS : out  STD_LOGIC;
    LCD_RW : out  STD_LOGIC;
    LCD_E : out  STD_LOGIC;
    -- ADD USER PORTS ABOVE THIS LINE ------------------

lcd.vhd: in architecture, instantiation of user_logic.vhd

      -- MAP USER PORTS BELOW THIS LINE ------------------
        LCD_data => LCD_data,
        LCD_RS => LCD_RS,
        LCD_RW => LCD_RW,
        LCD_E => LCD_E,
      -- MAP USER PORTS ABOVE THIS LINE ------------------

user_logic.vhd: in entity declaration

    -- ADD USER PORTS BELOW THIS LINE ------------------
    LCD_data : out  STD_LOGIC_VECTOR (3 downto 0);
    LCD_RS : out  STD_LOGIC;
    LCD_RW : out  STD_LOGIC;
    LCD_E : out  STD_LOGIC;
    -- ADD USER PORTS ABOVE THIS LINE ------------------
```

So that's the ports to the outside world. The only thing remaining is to instantiate an 8-bit driver and link it to the FIFO. The code to do this takes the form of yet another small FSM. Here's the architecture of user_logic that was written to do this (completely replacing the almost-empty architecture section in the generated code):

```
architecture IMP of user_logic is

    -- Signals and shared variables
    type STATE_TYPE is ( READY, LOADING, WAITING );
    signal state : STATE_TYPE := WAITING;

    signal data_store : STD_LOGIC_VECTOR(7 downto 0) :=
(others=>'0');
    signal RS_store, load : STD_LOGIC := '0';
    signal ack_busy : STD_LOGIC;

    -- External components
    component Driver_8Bit
        generic (
                CLK_MHZ : NATURAL
            );
        port (
                clk : in STD_LOGIC;
                data_in : in  STD_LOGIC_VECTOR (7 downto 0);
                RS_in : in  STD_LOGIC;
```

```
                load : in STD_LOGIC;
                ack_busy : out  STD_LOGIC;
                LCD_data : out  STD_LOGIC_VECTOR (3 downto 0);
                LCD_RS : out  STD_LOGIC;
                LCD_RW : out  STD_LOGIC;
                LCD_E : out  STD_LOGIC
            );
    end component;

begin

-- Driver module instantiation
driver : Driver_8Bit
    generic map (
        CLK_MHZ => 125
    )
    port map (
        clk => Bus2IP_Clk,
        data_in => data_store,
        RS_in => RS_store,
        load => load,
        ack_busy => ack_busy,
        LCD_data => LCD_data,
        LCD_RS => LCD_RS,
        LCD_RW => LCD_RW,
        LCD_E => LCD_E
    );

STATE_PROC: process (Bus2IP_Clk) is
begin
    if rising_edge(Bus2IP_Clk) then
        IP2WFIFO_RdReq <= '0';
        case state is
            when READY =>
                if WFIFO2IP_RdAck = '1' then
                    -- Valid data to read from fifo
                    data_store <= WFIFO2IP_Data(24 to 31);
                    RS_store <= WFIFO2IP_Data(23);
                    state <= LOADING;
                else
                    -- No valid data present yet
                    IP2WFIFO_RdReq <= '1';
                end if;
            when LOADING =>
                if ack_busy = '1' then
                    -- Load request acknowledged
                    state <= WAITING;
```

```
                end if;
            when WAITING =>
                if ack_busy = '0' then
                    -- Driver free
                    state <= READY;
                end if;
        end case;
    end if;
end process;

with state select
    load <=    '1' when LOADING,
               '0' when others;

  ----------------------------------------
  -- Not using direct transfer of data
  ----------------------------------------
  IP2Bus_Data  <= (others => '0');

  IP2Bus_WrAck <= '0';
  IP2Bus_RdAck <= '0';
  IP2Bus_Error <= '0';

end IMP;
```

Once again it's just a simple three-state Moore machine, which waits for the driver to be ready, asserts data, and then waits for a response. The transition from the READY state to the LOADING state, however, depends on the status of the WFIFO2IP_RdAck signal. This is asserted by the FIFO in response to a logic high on IP2WFIFO_RdReq if there is data to be read. Hence, in the READY state, the IP2WFIFO_RdReq signal is asserted until a '1' is read on WFIFO2IP_RdAck, at which point the data is latched and the state transition occurs.

Notice also that the logic vectors in the Xilinx generated code are 'the wrong way round' - bit 31 is the LSB. That's why the data lines are mapped as they are, with lines 24 to 31 connected to the data lines and line 23 acting as the RS line.

The PAO file must also be modified, to include the 4-bit and 8-bit drivers:

```
##############################################################################
## Filename:          /media/windata/data/LCD_system1/pcores/
lcd_v1_00_a/data/lcd_v2_1_0.pao
## Description:       Peripheral Analysis Order
## Date:              Mon Nov  2 17:09:20 2009 (by Create and
Import Peripheral Wizard)
##############################################################################

lib proc_common_v2_00_a proc_common_pkg vhdl
```

```
lib proc_common_v2_00_a ipif_pkg vhdl
lib proc_common_v2_00_a or_muxcy vhdl
lib proc_common_v2_00_a or_gate128 vhdl
lib proc_common_v2_00_a family_support vhdl
lib proc_common_v2_00_a pselect_f vhdl
lib proc_common_v2_00_a counter_f vhdl
lib plbv46_slave_single_v1_00_a plb_address_decoder vhdl
lib plbv46_slave_single_v1_00_a plb_slave_attachment vhdl
lib plbv46_slave_single_v1_00_a plbv46_slave_single vhdl
lib proc_common_v2_00_a inferred_lut4 vhdl
lib proc_common_v2_00_a pf_counter_bit vhdl
lib proc_common_v2_00_a pf_counter vhdl
lib proc_common_v2_00_a pf_counter_top vhdl
lib proc_common_v2_00_a pf_occ_counter vhdl
lib proc_common_v2_00_a pf_occ_counter_top vhdl
lib proc_common_v2_00_a pf_adder_bit vhdl
lib proc_common_v2_00_a pf_adder vhdl
lib proc_common_v2_00_a pf_dpram_select vhdl
lib proc_common_v2_00_a srl16_fifo vhdl
lib dre_v2_00_a dsp48_mux vhdl
lib dre_v2_00_a tx_dre_top vhdl
lib wrpfifo_v4_00_a pf_dly1_mux vhdl
lib wrpfifo_v4_00_a wrpfifo_dp_cntl vhdl
lib wrpfifo_v4_00_a ipif_control_wr_dre vhdl
lib wrpfifo_v4_00_a wrpfifo_top vhdl
lib lcd_v1_00_b Driver_4Bit vhdl
lib lcd_v1_00_b Driver_8Bit vhdl
lib lcd_v1_00_b user_logic vhdl
lib lcd_v1_00_b lcd vhdl
```

The new lines are in bold. Note the order (reverse hierarchical).

Following an import of the new peripheral, as described above, the following lines need adding to the system UCF file:

```
NET  LCD_data_pin<0>        LOC="T9";
NET  LCD_data_pin<1>        LOC="G7";
NET  LCD_data_pin<2>        LOC="G6";
NET  LCD_data_pin<3>        LOC="T11";
NET  LCD_RW_pin             LOC="AC10";
NET  LCD_RS_pin             LOC="J17";
NET  LCD_E_pin              LOC="AC9";
```

These pin numbers were obtained from the XUPv5 schematic, but they could also have been gleaned from the master UCF, available from the Xilinx site.

Now for the C. Xilinx don't have any documentation on this, as far as I can tell. From other demos on the web, I'd ascertained that generating a peripheral also generates an include file with the same name - so 'lcd.h' in this case - which can be found in [project]\drivers\[peripheral name]\src. Looking in that reveals a bunch of useful functions

(most actually implemented as macros) for the peripheral. All of them take the peripheral base address as an argument, which is great if you know it, not if you don't; it's held in a constant called XPAR_LCD_0_BASEADDR for the first instance of a peripheral called 'lcd', so in the code below it's assigned to a global (eugh!) at the start of execution to keep the syntax neat. Calls to xil_printf() push text to the RS232 port, which is useful for debugging. Hopefully the C is self-explanatory, basically it just wraps calls to LCD_mWriteToFIFO() to push stuff to the LCD.

```
#include "xparameters.h"
#include "xbasic_types.h"
#include "xstatus.h"
#include "lcd.h"

/************************************************/
/* Constants                                    */
/************************************************/

// Display feature flags
#define LCD_DISPLAY_ON          0x4
#define LCD_DISPLAY_OFF         0x0
#define LCD_CURSOR_ON           0x2
#define LCD_CURSOR_OFF          0x0
#define LCD_CURSOR_BLINK        0x1
#define LCD_CURSOR_NO_BLINK     0x0

// Entry modes
#define LCD_ENTRY_CURSOR_RIGHT  0x2
#define LCD_ENTRY_CURSOR_LEFT   0x0
#define LCD_ENTRY_SCROLL_RIGHT  0x1
#define LCD_ENTRY_SCROLL_LEFT   0x3

// Cursor nudge
#define LCD_NUDGE_RIGHT         0x4
#define LCD_NUDGE_LEFT          0x0

// Scroll display
#define LCD_SCROLL_RIGHT        0x4
#define LCD_SCROLL_LEFT         0x0


/************************************************/
/* Global (device base address                  */
/************************************************/

Xuint32 baseaddr;


/************************************************/
/* Utility functions                            */
/************************************************/
```

```
void lcd_clear() {
    // Clears the screen and sets cursor to (0,0)
    LCD_mWriteToFIFO(baseaddr, 0, 0x01);
}

void lcd_home() {
    // Sets cursor to (0,0)
    LCD_mWriteToFIFO(baseaddr, 0, 0x00);
}

void lcd_entry_mode(Xuint32 command) {
    // Sets the entry mode (see LCD_ENTRY_* constants)
    LCD_mWriteToFIFO(baseaddr, 0, 0x04 | (command & 0x03));
}

void lcd_display_ctrl(Xuint32 command) {
    // Controls display and cursor (see LCD_DISPLAY_* and
LCD_CURSOR_* constants)
    LCD_mWriteToFIFO(baseaddr, 0, 0x08 | (command & 0x07));
}

void lcd_nudge_cursor(Xuint32 command) {
    // Nudges cursor right or left (see LCD_NUDGE_* constants)
    LCD_mWriteToFIFO(baseaddr, 0, 0x10 | (command & 0x04));
}

void lcd_scroll(Xuint32 command) {
    // Scrolls entire display (see LCD_SCROLL_* constants)
    LCD_mWriteToFIFO(baseaddr, 0, 0x18 | (command & 0x04));
}

void lcd_cgram_set(Xuint32 address) {
    // Sets entry point to a CGRAM location, allowing custom
character generation
    LCD_mWriteToFIFO(baseaddr, 0, 0x40 | (address & 0x3F));
}

void lcd_ddram_set(Xuint32 address) {
    // Sets entry point to a DDRAM location, allowing text entry
    LCD_mWriteToFIFO(baseaddr, 0, 0x80 | (address & 0x7F));
}

void lcd_data_write(Xuint32 data) {
    // Writes a byte to the current address (CGRAM or DDRAM)
    LCD_mWriteToFIFO(baseaddr, 0, 0x100 | (data & 0xFF));
}

/************************************************/
/* Convenience functions                        */
/************************************************/

void lcd_init() {
```

```
    // 4-bit mode, 2 line display, etc.
    LCD_mWriteToFIFO(baseaddr, 0, 0x2C);
    // Display on, no cursor
    lcd_display_ctrl(LCD_DISPLAY_ON);
    // Sensible default entry mode
    lcd_entry_mode(LCD_ENTRY_CURSOR_RIGHT);
    // Clear the screen
    lcd_clear();
}

void lcd_move(Xuint32 line, Xuint32 pos) {
    // Moves the cursor to a new position, line and pos both
0-indexed
    // Line may be 0 or 1, pos may be 0 to 39
    lcd_ddram_set((0x40 * (line%2)) + (pos%40));
}

void lcd_print(char *text) {
    // Prints a string to the screen (if the current address is in
DDRAM)
    while (*text != 0) {
        lcd_data_write(*text);
        ++text;
    }
}

/************************************************/
/* Main program                                 */
/************************************************/

int main (void) {
  Xuint32 i;
  Xuint32 temp;

  // Clear the (TTY) screen
  xil_printf("%c[2J",27);

  // Check that the peripheral exists
  baseaddr = (Xuint32)XPAR_LCD_0_BASEADDR;
  XASSERT_NONVOID(((Xuint32 *)baseaddr) != XNULL);

  xil_printf("LCD Test\n");

  // Reset write FIFO to initial state
  LCD_mResetWriteFIFO(baseaddr);

  // Push data to write FIFO
    lcd_init();
    lcd_print("Hello, world!");
    lcd_move(1,20);
    lcd_print("Second line...");
```

```
    xil_printf("Scrolling left...\n");

    Xuint32 count, loop;
    for (loop = 0; loop < 40; ++loop) {
        lcd_scroll(LCD_SCROLL_LEFT);
        count=0x2FFFFF;
        while (count--) asm volatile ("nop");
    }

    xil_printf("Blinking...\n");

    for (loop = 0; loop < 5; ++loop) {
        lcd_display_ctrl(LCD_DISPLAY_OFF);
        count=0x4FFFFF;
        while (count--) asm volatile ("nop");
        lcd_display_ctrl(LCD_DISPLAY_ON);
        count=0x4FFFFF;
        while (count--) asm volatile ("nop");
    }

    xil_printf("Scrolling right...\n");

    for (loop = 0; loop < 40; ++loop) {
        lcd_scroll(LCD_SCROLL_RIGHT);
        count=0x2FFFFF;
        while (count--) asm volatile ("nop");
    }

    xil_printf("End of test\n\n");

    // Stay in an infinite loop
    while(1);

}
```

Note that the C still depends on looping wait functions to implement the scrolling and blinking. A nice extension would be to create a programmable timer peripheral with an interrupt line, and then hang an ISR off it which could handle scrolling and blinking functions.

Controlling the Screen: PicoBlaze Solution

The VHDL required for the above could no doubt be simplified a little, given time and ingenuity. But still, the number of state machines in there suggests strongly that driving the LCD would be more easily done in software (indeed, the simplicity of the 'software solution' above confirms that). Enter the PicoBlaze. It's a small 8-bit microprocessor core with basic GPIOs and interrupt service functionality. It's a bit like a PIC, especially in that it's a Harvard architecture device (instructions and data are completely separate). It has 16 general purpose registers, with 64 bytes of 'scratchpad RAM' available for temporary

storage if 16 registers isn't enough. It has a separate, managed, 31-location call stack. Every instruction takes two clock cycles to execute.

Interestingly, Xilinx have a little section in the PicoBlaze user guide on why you might want to embed a CPU in programmable hardware. Here it is, verbatim:

Microcontrollers and FPGAs both successfully implement practically any digital logic function. However, each has unique advantages in cost, performance, and ease of use. Microcontrollers are well suited to control applications, especially with widely changing requirements. The FPGA resources required to implement the microcontroller are relatively constant. The same FPGA logic is re-used by the various microcontroller instructions, conserving resources. The program memory requirements grow with increasing complexity.

Programming control sequences or state machines in assembly code is often easier than creating similar structures in FPGA logic.

Microcontrollers are typically limited by performance. Each instruction executes sequentially. As an application increases in complexity, the number of instructions required to implement the application grows and system performance decreases accordingly. By contrast, performance in an FPGA is more flexible. For example, an algorithm can be implemented sequentially or completely in parallel, depending on the performance requirements. A completely parallel implementation is faster but consumes more FPGA resources.

A microcontroller embedded within the FPGA provides the best of both worlds. The microcontroller implements non-timing crucial complex control functions while timing-critical or data path functions are best implemented using FPGA logic. For example, a microcontroller cannot respond to events much faster than a few microseconds. The FPGA logic can respond to multiple, simultaneous events in just a few to tens of nanoseconds. Conversely, a microcontroller is cost-effective and simple for performing format or protocol conversions.

There's also a little summary table:

	PicoBlaze Microcontroller	FPGA Logic
Strengths	• Easy to program, excellent for control and state machine applications • Resource requirements remain constant with increasing complexity • Re-uses logic resources, excellent for lower-performance functions	• Significantly higher performance • Excellent at parallel operations • Sequential vs. parallel implementation trade-offs optimise performance or cost • Fast response to multiple, simultaneous inputs
Weaknesses	• Executes sequentially • Performance degrades with increasing complexity	• Control and state machine applications more difficult to program

	• Program memory requirements increase with increasing complexity • Slower response to simultaneous inputs	• Logic resources grow with increasing complexity

PicoBlaze Introduction

For the Virtex 5, download the version for the Virtex II Pro / Spartan 3 / Virtex 4 (there isn't a specific Virtex 5 version yet). You'll get a zip with loads of things in it! The most useful bits are:

- The VHDL folder, specifically kcpsm3.vhd and embedded_kcpsm3.vhd
- The assembler, KCPSM3.EXE, in the Assembler folder
- The ROM template, ROM_form.vhd, also in the Assembler folder

Note that I have **not used** the Windows command-line assembler, nor the free pBlazeIDE software from http://www.mediatronix.com/pBlazeIDE.htm that's recommended by Xilinx in the PicoBlaze manual. The latter has stepwise simulation and debugging, and might be a more suitable choice in a learning environment than the command-line assembler. The only thing I've used is kpicosim under Linux, which I believe to be similar to the Mediatronix software.

The kcpsm3.vhd file contains the complete PicoBlaze processor. Its connections are as follows:

- OUT: ADDRESS[9:0]
- IN: INSTRUCTION[17:0]
- IN: IN_PORT[7:0]
- OUT: OUT_PORT[7:0]
- OUT: PORT_ID[7:0]
- OUT: READ_STROBE
- OUT: WRITE_STROBE
- IN: INTERRUPT
- OUT: INTERRUPT_ACK
- IN: CLK
- IN: RESET

The functions of these connections will now be described (except for CLK and RESET, which are kind of obvious)!

The program storage ROM is connected to the ADDRESS outputs and the INSTRUCTION inputs, which act as address and data lines respectively. Instructions are 16 bits wide but have two parity bits, making full use of the 18-bit width of a BRAM (which is the most efficient way to store the program data, apparently).

The GPIO mechanism is quite clever. There are notionally 256 input ports and 256 output ports. When the PicoBlaze initiates a read from a GPIO port (in response to an INPUT command), the PORT_ID output is loaded with the number of the input port in question, and then the READ_STROBE line is strobed. It is up to the user to ensure that when the line is strobed, the correct data exists on the IN_PORT input. Similarly, when the PicoBlaze initiates a GPIO write in response to an OUTPUT command, the PORT_ID output is loaded

with the number of the output port in question, and the WRITE_STROBE line is strobed. Again it is up to the user to ensure that the data on the OUT_PORT output is latched somewhere as appropriate. The READ_STROBE line doesn't seem to be of much use, because the value of the IN_PORT input is ignored unless a read is taking place; a simple (combinational) selected assignment based on the value of PORT_ID is therefore sufficient to drive IN_PORT. The WRITE_STROBE is vital though, as it can be used to latch valid data from OUT_PORT when a write is performed. An example will be included later. Note that the Xilinx recommendation is that, for 2-8 inputs or outputs, 'one-high' encoding should be used (in other words, notional ports 0,1,2,4,8,16,32,64 should be used) so that the lines of PORT_ID can be used directly as enable inputs to multiplexers and demultiplexers. For more than 8 inputs or outputs, binary encoding must be used, but the multiplexers and demultiplexers should be connected in a cascade, each connected to exactly one bit of PORT_ID. This will ensure maximum performance.

INTERRUPT (an input) and INTERRUPT_ACK (an output) are, logically enough, used for interrupt handling. I've never used them. When INTERRUPT is driven high, the processor will execute a CALL 0x3FF instruction (a call to the last instruction in memory) and assert the INTERRUPT_ACK signal as soon as it has done so. The contents of instruction location 0x3FF should be a jump instruction to an ISR. For more details see the PicoBlaze user guide. INTERRUPT should be tied low if not required.

The embedded_kcpsm3.vhd file is a simple wrapper for the PicoBlaze which instantiates a kcpsm3 along with a component for storing the program code, and connects up the ADDRESS and INSTRUCTION lines, exposing the remainder of the PicoBlaze's connections through the wrapper. By default, the component used for storing program code is called 'prog_rom'. Hence, unless this file is modified, the VHDL file containing the initialised program ROM should contain an entity called 'prog_rom', which probably instantiates a BRAM.

Because the PicoBlaze is just a VHDL description of a simple processor, it can easily be implemented in a straight VHDL design in the ISE. This might provide a good way in to the idea of implementing an embedded system on an FPGA, before engaging with the forest of wizards and compilers that comprise the EDK..

Programming the PicoBlaze

The easiest way to program the PicoBlaze is by using a template (such as the aforementioned 'ROM_form.vhd') which can be given to a suitable assembler. The assembler will, having assembled the code, fill in the gaps in the template in order to initialise the ROM with the program data. The embedded_kcpsm3.vhd module can then be synthesised. This method of programming is good for teaching purposes I think - it's very clear what the assembler is doing, you can open up the finished VHDL file and see the initialised ROM contents, etc. However, it has one big disadvantage: changing the code means changing the initialisation of the ROM, which means invalidating the synthesis for the ROM, which means having to resynthesise the entire design for *any* code change. Not fast.

There is an alternative, which is to manipulate the synthesised bitstream directly to change the contents of the program ROM. Direct manipulation of most parts of the synthesised bitstream would be asking for trouble, but just changing the initialisation values of a BRAM doesn't affect the validity of the synthesis. I googled around, and the clearest page I found

on this process was Andy Greensted's, on his labbookpages.co.uk. He's written a script to do it, but it's for Linux; I'm not sure if the idea would be easily portable, because I haven't looked that thoroughly. Nor, in fact, have I even tried this process - I've made good use of simulation instead, to ensure the code works before it's tested on the device!

LCD Control on the PicoBlaze

I'm going to dive straight in here and present my assembler code. It's mostly comments! There are under 100 actual instructions here. The longest section is the big sets of timing loops towards the end. For various reasons I've tried this design clocked at a variety of speeds, and so my timing loops have had to be different lengths - there are versions for 50, 100 and 125MHz here, and currently all but the 100MHz ones are commented out (the PLB runs at 100MHz in my design).

```
    ; Registers s0-s7 are reserved for main program flow
    ; (only s0 is currently used)
    ; Registers s8-sA are reserved for the timer loops
    ; Registers sB-sD are currently unreserved
    ; Registers sE and sF are reserved for input data storage
    namereg sE, data_in
    namereg sF, control_in

    constant rs_in,       0x01
    constant load_in,     0x02
    constant busy_out,    0x01

    constant lcd_rw,      0x10
    constant lcd_rs,      0x20
    constant lcd_e,       0x40
    constant lcd_mask,    0x2F    ; masks out all but data and RS

; Every instruction is 16ns at 125MHz (at 1/2 clock)
; Every instruction is 20ns at 100MHz (at 1/2 clock)
; Every instruction is 40ns at 50MHz (at 1/2 clock)

    address 000

INIT:
    load s0, busy_out
    output s0, 1    ; Set busy flag
    load s0, 0x00
    output s0, 0    ; Clear LCD outputs
    load data_in, 0
    load control_in, 0    ; Clear the 'read' data (there is none!)

    call WAIT_50ms

    load s0, 0x02
    call PUSH_4BIT
    call PUSH_4BIT
```

```
    call PUSH_4BIT    ; Force 4-bit mode

    call WAIT_15ms

    load s0, 0x02
    call PUSH_4BIT
    load s0, 0xC0
    call PUSH_4BIT    ; Display setup

    call WAIT_15ms

MAIN_LOOP:
    ; Using a call here is a hack to save logic - it allows the
main subroutine
    ; to make use of the 'return' statements at the end of the
delay blocks
    call MAIN_SUBROUTINE
    jump MAIN_LOOP

MAIN_SUBROUTINE:
STATE_IDLE:
IDLE_WAIT_FOR_LOAD_LOW:
    input control_in, 1
    test control_in, load_in
    jump NZ, IDLE_WAIT_FOR_LOAD_LOW
    load s0, 0x00
    output s0, 1        ; Clear busy flag
IDLE_WAIT_FOR_LOAD_HIGH:
    input control_in, 1
    test control_in, load_in
    jump Z, IDLE_WAIT_FOR_LOAD_HIGH
STATE_ACTIVE:
    input data_in, 0    ; control_in already loaded
    load s0, busy_out
    output s0, 1    ; Set busy flag
    load s0, data_in
    sr0 s0
    sr0 s0
    sr0 s0
    sr0 s0            ; Find ms-nibble of data
    test control_in, rs_in
    jump Z, ACTIVE_J1
    or s0, lcd_rs        ; Add RS flag to output byte if set
ACTIVE_J1:
    call PUSH_4BIT        ; Push ms-nibble to LCD
    load s0, data_in
    and s0, 0x0F        ; Find ls-nibble of data
    test control_in, rs_in
    jump Z, ACTIVE_J2
    or s0, lcd_rs        ; Add RS flag to output byte if set
ACTIVE_J2:
    call PUSH_4BIT        ; Push ls-nibble to LCD
```

```
    ; Now must wait for a while, the time depends on the
instruction
    ; Using 'jump' to call a delay subroutine means that the
'return'
    ; instruction at the end of the chosen delay subroutine will in
fact
    ; return from _this_ subroutine to MAIN_LOOP
    test control_in, rs_in
    jump NZ, WAIT_44us    ; Write operations need a 43us min delay
    test data_in, 0xFC
    jump NZ, WAIT_44us    ; Config operations need a 39us min delay
    jump WAIT_2ms        ; Clear or home operations need a 1.53ms
min delay
                ; (which experimentation has shown not to be long
enough!)

PUSH_4BIT:
    and s0, lcd_mask    ; Unset RW and E lines
    output s0, 0        ; Push to LCD
    load s0, s0        ; nop: Assert for 60ns (<4 instructions)
    load s0, s0        ; nop
    or s0, lcd_e        ; LCD load line active
    output s0, 0        ; Push to LCD
    call WAIT_450ns        ; Assert for 450ns
    load s0, s0        ; nop (above is really 448ns, this is for
safety!)
    xor s0, lcd_e        ; LCD load line inactive
    output s0, 0        ; Push to LCD
    call WAIT_450ns        ; Assert for 450ns
    return

WAIT_50ms:
    ; At 125MHz:
    ; 50ms needs 3125000 instructions; with 8 overhead
    ; (inc call and return) that's 3124992, so
    ; 65536*s8 + 256*s9 + sA + 1 = 781248 (for 24-bit loop)
    ; so s8=11=0x0B, s9=235=0xEB, sA=191=0xBF
;    load s8, 0x0B
;    load s9, 0xEB
;    load sA, 0xBF
    ; At 100MHz:
    ; 50ms needs 2500000 instructions; with 8 overhead
    ; (inc call and return) that's 2499992, so
    ; 65536*s8 + 256*s9 + sA + 1 = 624998 (for 24-bit loop)
    ; so s8=9, s9=137=0x89, sA=101=0x65
    load s8, 0x09
    load s9, 0x89
    load sA, 0x65
    ; At 50MHz:
    ; 50ms needs 1250000 instructions; with 8 overhead
    ; (inc call and return) that's 1249992, so
    ; 65536*s8 + 256*s9 + sA + 1 = 312498 (for 24-bit loop)
```

```
    ; so s8=4, s9=196=0xC4, sA=177=0xB1
;    load s8, 0x04
;    load s9, 0xC4
;    load sA, 0xB1

    load s0, s0    ; nop
    load s0, s0    ; nop
    jump WAIT_LOOP_24bit
WAIT_15ms:
    ; At 125MHz:
    ; 15ms needs 937500 instructions; with 8 overhead
    ; (inc call and return) that's 937492, so
    ; 65536*s8 + 256*s9 + sA + 1 = 234373 (for 24-bit loop)
    ; so s8=3, s9=147=0x93, sA=132=0x84
;    load s8, 0x03
;    load s9, 0x93
;    load sA, 0x84
    ; At 100MHz:
    ; 15ms needs 750000 instructions; with 8 overhead
    ; (inc call and return) that's 749995, so
    ; 65536*s8 + 256*s9 + sA + 1 = 187498 (for 24-bit loop)
    ; so s8=2, s9=220=0xDC, sA=105=0x69
    load s8, 0x02
    load s9, 0xDC
    load sA, 0x69
    ; At 50MHz:
    ; 15ms needs 375000 instructions; with 8 overhead
    ; (inc call and return) that's 374992, so
    ; 65536*s8 + 256*s9 + sA + 1 = 93748 (for 24-bit loop)
    ; so s8=1, s9=110=0x6E, sA=51=0x33
;    load s8, 0x01
;    load s9, 0x6E
;    load sA, 0x33

    load s0, s0    ; nop
    load s0, s0    ; nop
    jump WAIT_LOOP_24bit    ; effectively a nop
WAIT_LOOP_24bit:
    sub sA, 1
    subcy s9, 0
    subcy s8, 0
    jump NC, WAIT_LOOP_24bit
    ; Number of instructions above is
    ; 4*(65536*s8 + 256*s9 + sA + 1)
    return

WAIT_2ms:
    ; At 125MHz:
    ; 2ms needs 125000 instructions; with 5 overhead
    ; (inc call and return) that's 124995, so
    ; 256*s8 + s9 + 1 = 41665 (for 16-bit loop)
    ; so s8=162=0xA2, s9=192=0xC0
```

```
;    load s8, 0xA2
;    load s9, 0xC0
    ; At 100MHz:
    ; 2ms needs 100000 instructions; with 7 overhead
    ; (inc call and return) that's 99993, so
    ; 256*s8 + s9 + 1 = 33331 (for 16-bit loop)
    ; so s8=130=0x82, s9=50=0x32
    load s8, 0x82
    load s9, 0x32
    load s0, s0    ; nop
    load s0, s0    ; nop
    ; At 50MHz:
    ; 2ms needs 50000 instructions; with 5 overhead
    ; (inc call and return) that's 49995, so
    ; 256*s8 + s9 + 1 = 16665 (for 16-bit loop)
    ; so s8=65=0x41, s9=24=0x18
;    load s8, 0x41
;    load s9, 0x18

    jump WAIT_LOOP_16bit
WAIT_44us:
    ; At 125MHz:
    ; 44us needs 2750 instructions; with 5 overhead
    ; (inc call and return) that's 2745, so
    ; 256*s8 + s9 + 1 = 915 (for 16-bit loop)
    ; so s8=3, s9=146=0x92
;    load s8, 0x03
;    load s9, 0x92
;    load s0, s0    ; nop
    ; At 100MHz:
    ; 44us needs 2200 instructions; with 4 overhead
    ; (inc call and return) that's 2196, so
    ; 256*s8 + s9 + 1 = 732 (for 16-bit loop)
    ; so s8=2, s9=219=0xDB
    load s8, 0x03
    load s9, 0xDB
    ; At 50MHz:
    ; 44us needs 1100 instructions; with 5 overhead
    ; (inc call and return) that's 1095, so
    ; 256*s8 + s9 + 1 = 365 (for 16-bit loop)
    ; so s8=1, s9=108=0x6C
;    load s8, 0x01
;    load s9, 0x6C
;    load s0, s0    ; nop

WAIT_LOOP_16bit:
    sub s9, 1
    subcy s8, 0
    jump NC, WAIT_LOOP_16bit
    ; Number of instructions above is
    ; 3*(256*s8 + s9 + 1)
    return
```

```
WAIT_450ns:
    ; At 125MHz:
    ; 450ns needs just over 28 instructions
    ; With 4 overhead (inc call and return) that's 12
    ; tight loops, or s8=11=0x0B.  Total delay 28*16ns=448ns
;    load s8, 0x0B
;    load s0, s0    ; nop
    ; At 100MHz:
    ; 450ns needs 22.5 instructions
    ; With 4 overhead (inc call and return) that's 9
    ; tight loops, or s8=8.  Total delay 22*20ns=440ns
    load s8, 0x0B
    load s0, s0    ; nop
    ; At 50MHz:
    ; 450ns needs 11.25 instructions
    ; With 3 overhead (inc call and return) that's 4
    ; tight loops, or s8=3.  Total delay 11*40ns=440ns
;    load s8, 0x03

WAIT_LOOP_8bit:
    sub s8, 1
    jump NC, WAIT_LOOP_8bit
    ; Number of instructions above is
    ; 2*(s8 + 1)
    return
```

The code is fairly simple in structure. The initialisation section (INIT) sets the 'ack_busy' line high, then performs the various writes and waits required to initialise the LCD. The main loop (MAIN_LOOP) then repeatedly calls MAIN_SUBROUTINE; this checks that the 'load' input is low, then deasserts 'ack_busy', waits for 'load' to become high, loads the 8-bit data and RS signals, and then pushes the two nibbles one at a time to the LCD, before waiting some time for the command to complete and returning. Both INIT and MAIN_SUBROUTINE call PUSH_4BIT, which pushes the least significant four bits of register s0 to the LCD, along with a copy of the 'RS' flag, and asserts/deasserts the LCD 'E' line appropriately. The remainder of the code is timing loops; there's an 8-bit loop for the 450ns wait, a 16-bit loop for the 44us and 2ms waits, and a 24-bit loop for the 15ms and 50ms waits. The hardest bit of coding these wait loops was calculating the initial register values to get the timings right!

Creating a Peripheral

The above was tested by modifying 'Feeder.vhd' from above to push data to the kcpsm3 unit instead of the Driver_8Bit unit. The changes required are minor so this is left as an 'exercise for the reader'! Similar modifications were required to the peripheral wrapper. If a peripheral called 'lcd_pico' is generated in a new EDK project, embedded_kcpsm3 can then be instantiated directly from within user_logic.vhd. The process of generating the peripheral, making connections to the outside world, and re-importing the modified peripheral is identical to last time. The only thing that has changed is the architecture section of user_logic:

```
architecture IMP of user_logic is

    -- Signals and shared variables
    type STATE_TYPE is ( READY, LOADING, WAITING );
    signal state : STATE_TYPE := WAITING;

    signal data_store : STD_LOGIC_VECTOR(7 downto 0) :=
(others=>'0');
    signal RS_store, load : STD_LOGIC := '0';
    signal ack_busy : STD_LOGIC := '1';

    -- Picoblaze signals
    signal port_id, pico_in, pico_out : std_logic_vector(7 downto
0);
    signal write_strobe : std_logic;

    -- External components
    component embedded_kcpsm3
    Port (
            port_id : out std_logic_vector(7 downto 0);
            write_strobe : out std_logic;
            read_strobe : out std_logic;
            out_port : out std_logic_vector(7 downto 0);
            in_port : in std_logic_vector(7 downto 0);
            interrupt : in std_logic;
            interrupt_ack : out std_logic;
            reset : in std_logic;
            clk : in std_logic
        );
    end component;

begin

-- Driver module instantiation
driver : embedded_kcpsm3
    port map (
        port_id => port_id,
        write_strobe => write_strobe,
--      read_strobe is unconnected,
        out_port => pico_out,
        in_port => pico_in,
        interrupt => '0',
--      interrupt_ack is unconnected,
        reset => '0',
        clk => pico_clk
    );

-- Picoblaze IO port mapping
-- Inputs:
with port_id(0) select
```

```
    pico_in <=    data_store when '0',
                    "000000" & load & RS_store when others;
-- Outputs:
PICO_WRITE_PROC: process(Bus2IP_Clk, write_strobe) is
begin
    if rising_edge(Bus2IP_Clk) then
        if write_strobe = '1' then
            case port_id(0) is
                when '0' =>
                    LCD_data <= pico_out(3 downto 0);
                    LCD_RW <= pico_out(4);
                    LCD_RS <= pico_out(5);
                    LCD_E <= pico_out(6);
                when '1' =>
                    ack_busy <= pico_out(0);
                when others =>
                    null;
            end case;
        end if;
    end if;
end process PICO_WRITE_PROC;

STATE_PROC: process (Bus2IP_Clk) is
begin
    if rising_edge(Bus2IP_Clk) then
        IP2WFIFO_RdReq <= '0';
        case state is
            when READY =>
                if WFIFO2IP_RdAck = '1' then
                    -- Valid data to read from fifo
                    data_store <= WFIFO2IP_Data(24 to 31);
                    RS_store <= WFIFO2IP_Data(23);
                    state <= LOADING;
                else
                    -- No valid data present yet
                    IP2WFIFO_RdReq <= '1';
                end if;
            when LOADING =>
                if ack_busy = '1' then
                    -- Load request acknowledged
                    state <= WAITING;
                end if;
            when WAITING =>
                if ack_busy = '0' then
                    -- Driver free
                    state <= READY;
                end if;
        end case;
    end if;
end process;

with state select
```

```
    load <=    '1' when LOADING,
               '0' when others;

  ----------------------------------------
  -- Not using direct transfer of data
  ----------------------------------------
  IP2Bus_Data  <= (others => '0');

  IP2Bus_WrAck <= '0';
  IP2Bus_RdAck <= '0';
  IP2Bus_Error <= '0';

end IMP;
```

Note that there's still a state machine, run by STATE_PROC, to handle the transfer of data from the FIFO to the PicoBlaze. There's also a PICO_WRITE_PROC process to latch the PicoBlaze outputs when the write strobe is asserted, and a selected assignment to feed the input port of the PicoBlaze. And that's just about it. The associated C code for this peripheral is identical to before, except that the periheral is called 'lcd_pico' rather than 'lcd' (and its instance is 'lcd_pico_0' rather than 'lcd_0'), so the names of the functions and constants have to change. But that's it, and this software-based (or is it firmware-based?) driver performs identically to the hardware version above.

VHDL notes

I have not been terribly careful with capitalisation standards in my code, though in general I've used capitals for type names and state names, and lower-case for commands, variables and signals. Where I've interfaced to stuff that has spewed from Xilinx wizards, I've stuck with their capitalisation. I have *not* used the evil 'ieee.std_logic_unsigned' package or its variants, and I intend to continue this practice; if I want to perform arithmetic operations on bit-based signals, I'll use the 'ieee.numeric_std' package, though all of the code here avoids the process entirely - it's perfectly possible to synthesise a counter, for example, using integer or natural types (bounded subtypes are even better). I've also mixed and matched component instantiation with direct entity instantiation - I tend to use the latter for compactness, but some of the Xilinx tools generate the former and I didn't bother changing them.