# Project Specification and Guidelines

## 1. INTRODUCTION

This module is intended to teach you how to design a complex electronic system for implementation as a semi-custom integrated circuit. In the course of this module you will gain practical experience in the use of the "Top-Down" design methodology using a hardware description language (HDL) and simulator for the design task, and autoplacement and autorouting tools for the layout task. You will also get the opportunity to practise the Finite-State Machine (FSM) design methods taught in the "Foundations of VLSI" module. Completed IC designs will be 'fabricated' on programmable CMOS logic arrays from Actel, and should be available for testing at the end of the module.

## 2. PROJECT SPECIFICATION

The electronic system you are asked to design is a low-frequency rate-meter (or frequency counter). This system will take a low frequency signal (in the range 0.5Hz to 5Hz), and use an LCD display to output a corresponding pulses-per-minute (ppm) or pulses-per-second (Hz) count as required, with an accuracy of ±1-digit on a 3.5-digit display.

At high input frequencies this task can be accomplished by comparing with another signal of a known-but-lower frequency. The unknown frequency signal is used to clock a counter, and the counter state at the end of one period of the known signal then gives an indication of the frequency ratio, and the unknown frequency can thus be derived and displayed. For example, if 1% resolution were required and if the unknown frequency were in the 1Hz range as in this project, then the known frequency would have to be of the order of 0.01Hz. This causes a problem in that the frequency display will only be updated once every 100 seconds! Furthermore, any variations in the unknown frequency would be averaged out over the 100 second period and lost. Another, faster, method is required.

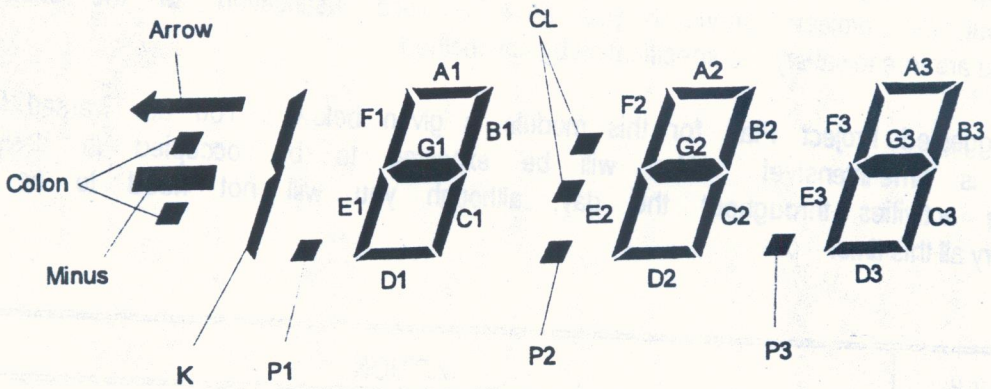
The alternative method which you are asked to use, is to drive the counter from the known signal (rather than the unknown), and to 'count' the length of the unknown periods. The count is then inversely proportional to the unknown frequency. By inverting this value the frequency can be obtained, with a new value being obtained once every period. Local variations in frequency will not be averaged out in this case.

If the number of cycles of the known frequency is N, then the unkown frequency can be calculated from fu=fo/N. Non-integer mathematics can be avoided simply by scaling by x100 before the calculation is done, and then scaling by 0.01 (ie shifting the decimal point) afterwards. The calculation then becomes:

$$fu = \{(fo.100)/N\}\{1/100\}$$

The system you will design thus requires two inputs for the unknown signal (fu) and the known signal (fo), derives the period of the unknown signal from fo, inverts the period to obtain the frequency, and then displays this frequency value. A third input is used to select between a pulses-per-minute or pulses-per-second value (ppm_hz). A reset signal should also be applied to allow initialisation of the state of the system. The output from the system is required to drive a 3½-digit LCD display (see next section).

## 3.LCD DATA SHEET




| PIN | SEGMENT | PIN | SEGMENT | PIN | SEGMENT | PIN | SEGMENT |
|---|---|---|---|---|---|---|---|
| 1 | Back-plane | 11 | C1 | 21 | A3 | 31 | F1 |
| 2 | Minus | 12 | P2 | 22 | F3 | 32 | G1 |
| 3 | K | 13 | E2 | 23 | G3 | 33 | |
| 4 | | 14 | D2 | 24 | B2 | 34 | |
| 5 | | 15 | C2 | 25 | A2 | 35 | |
| 6 | | 16 | P3 | 26 | F2 | 36 | |
| 7 | | 17 | E3 | 27 | G2 | 37 | |
| 8 | P1 | 18 | D3 | 28 | CL | 38 | Arrow |
| 9 | E1 | 19 | C3 | 29 | B1 | 39 | Colon |
| 10 | D1 | 20 | B3 | 30 | A1 | 40 | Back-plane |

AC drive: 4 - 12 Vrms (5 - 8 Vrms typical),
25-75 Hz (32 Hz typical)

DC component: < 50 mV

## 4. PROJECT PLAN

These notes include possible structures for modelling the specified system. It should be emphasised that these are suggestions only, and you are encouraged to come up with your own solution. There are many design decisions to be made throughout the project: provided you have a valid justification for the choices you make you are free to satisfy the specification by any method.

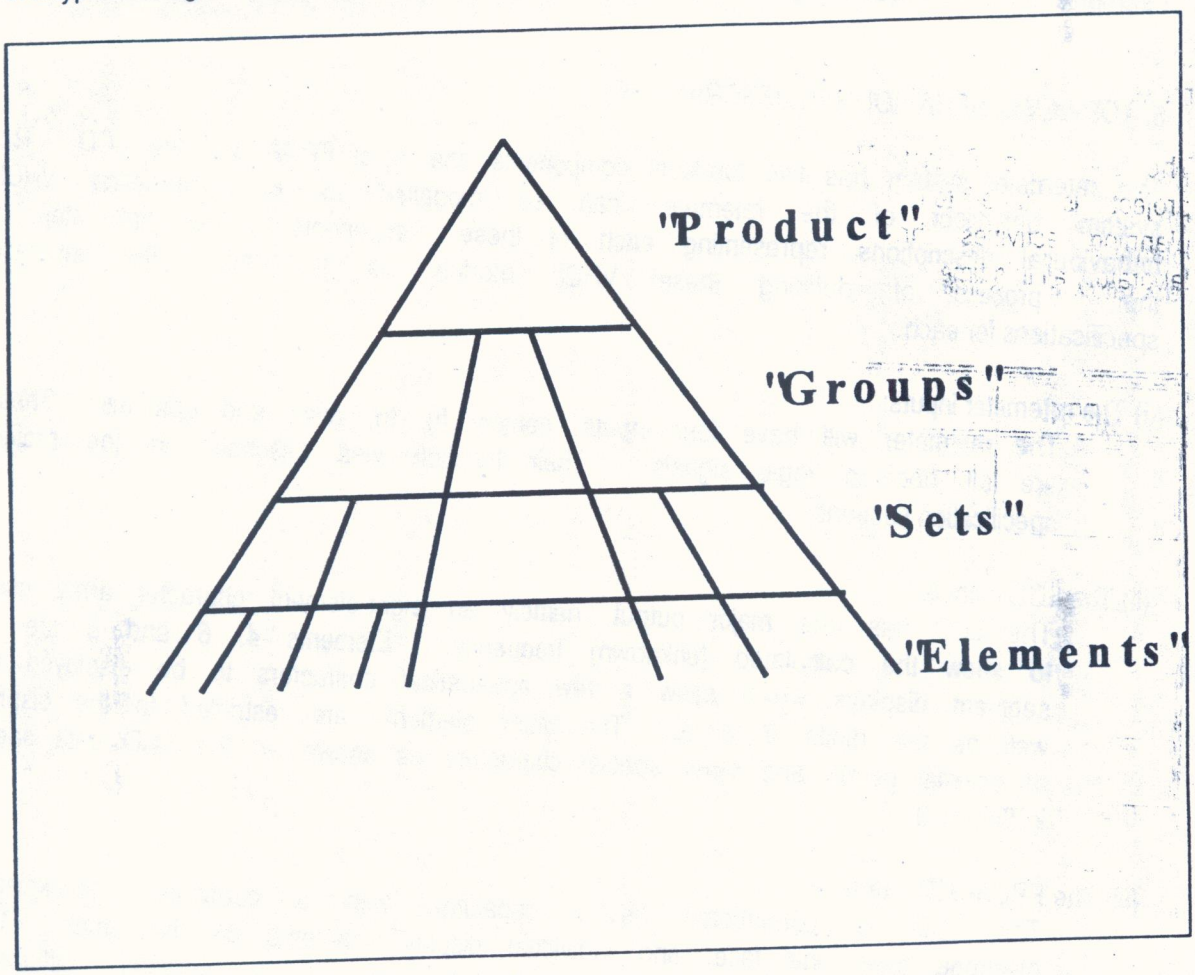
The suggested Project Plan for this module is given below. You are warned that this project is time-intensive! You will be expected to be occupied in design and planning activities throughout the day, although you will not need to be in the laboratory all this time.

| WEEK | ACTION |
|---|---|
| 1 | Introduction to OpenWindows & Sun workstations.<br>Introduction to Mentor-Graphics Design Architect & QuickSim |
| 2 | Introduction to VHDL.<br>Define VHDL top-level behavioural descriptions for FPGA and LCD.<br>Draw sheet for rate-meter, simulate & validate design. |
| 3 | Define VHDL behavioural descriptions for FPGA building blocks - measure, calculate and lcd_driver. |
| 4 | Draw sheet for FPGA, simulate & validate design. |
| 5 | Define VHDL control algorithms and Register-Transfer Logic for building blocks. |
| 6 | |
| 7 | Define gate-level logic (schematic entry using Actel cells) |
| 8 | |
| 9 | Place & Route Actel FPGA.<br>Post-layout simulation. |
| 10 | Program FPGA and Test.<br>Write Final Report. |

## 5. TOP-DOWN DESIGN METHODOLOGY

The top-down design methodology which you are required to follow in this project, essentially relies upon the repeated decomposition of a design "problem" into simpler components, until the lowest-level component is compatible with the building blocks available from the chosen technology (eg logic gates or cells from the technology library). At the higher levels, the components will be described

by VHDL behavioural descriptions. As the design proceeds, these will be replaced by schematic diagrams showing structures built up from lower-level components, which may themselves be described by VHDL or other schematics. The decomposition of a typical design is illustrated in the following figure:




In performing the design decomposition, you should note that the I/O specification between components on the same level remains unchanged at lower levels. This is important as it allows several advantages, namely:

(i) the "test-data" used to validate the design remains constant as the design develops (the timing of output signals may change, but their value should not).

(ii) Large design projects can be worked on by teams, with each team only concerned with meeting the I/O specification of "their" components. How other teams choose to implement other components should not affect the system operation.

(iii) Designs can be developed "piecemeal", ie components can be individually taken down to gate-level while the whole system is simulated as a combination of mixed-levels.

To be compatible with the lowest design level, which is generally composed of boolean logic, the I/O specification must be in terms of binary data signals. At the higher levels however, it may be more convenient to describe component

behaviour using integer or other discrete signal values. These dual requirements can be met through the use of functions or procedures to convert between the internal (behavioural) signal types and the interface binary signal types. Examples of this technique are given in the folowing sections.

## 6. TOP-LEVEL BEHAVIOURAL DESCRIPTION

The ratemeter system has two top-level components: the Actel FPGA and the LCD. The system behaviour of the ratemeter can be modelled by two interacting VHDL behavioural descriptions representing each of these components. The first step in the process of defining these VHDL models, is to define the interface specifications for each.

(i) The ratemeter inputs:

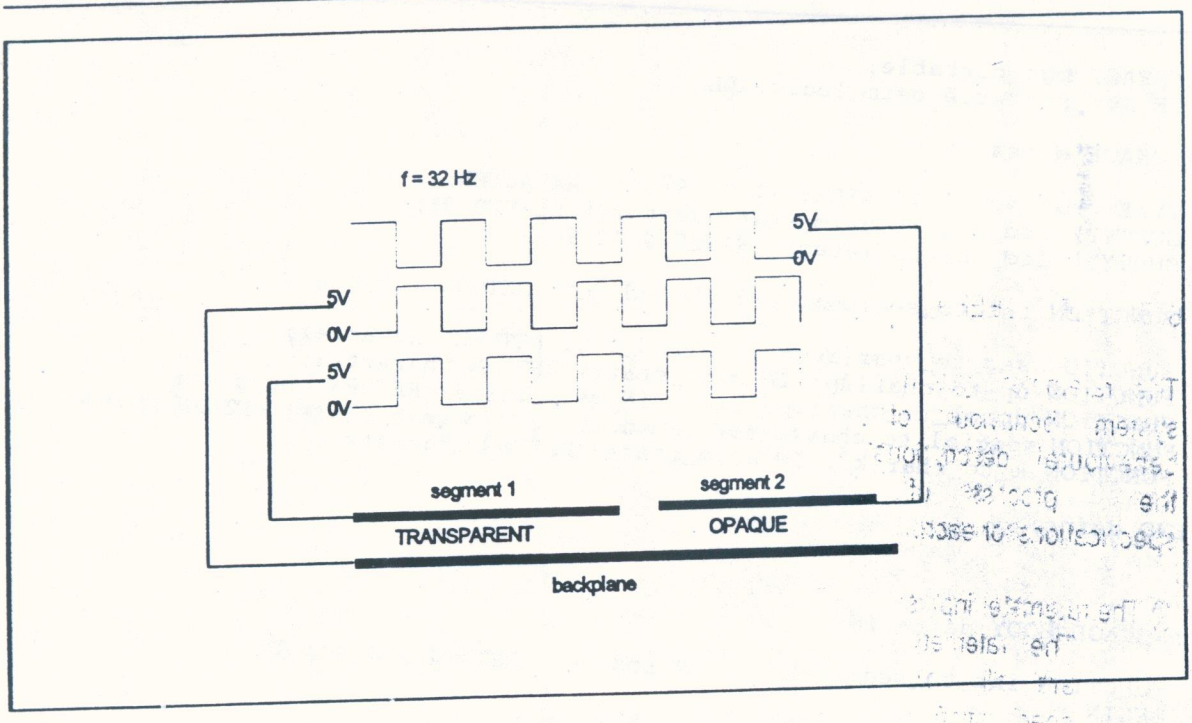
The ratemeter will have four inputs, namely fu, fo, reset and ppm_hz. These are all boolean logic signals. Their function was described in the project specification section.

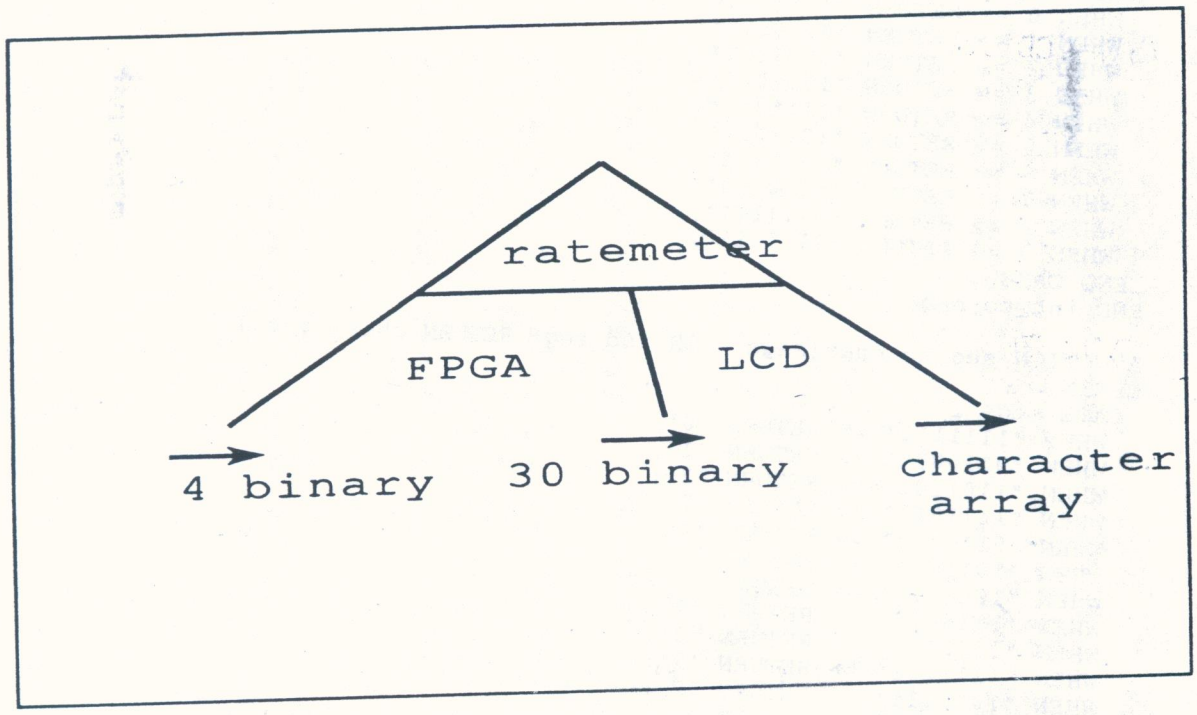
(ii) The LCD outputs:

The LCD has one major output, namely an eight-element character array used to show the calculated (unknown) frequency. Elements 4, 6 and 8 are 7-segment displays which allow a few aphabetical characters to be displayed as well as the digits 0 to 9. The other elements are restricted to the display of decimal points and other special characters as shown on the LCD data sheet in Section 3.

(iii) The FPGA-LCD interface:

The LCD is constructed as a capacitor, with a continuous (backplane) electrode over one face, and individual display segments on the other. The liquid crystal is normally transparent, however if a potential difference is imposed between a segment and the backplane, then the intervening liquid crystal will become opaque. For maximum lifetime, the LCD specification requires ac drive signals of 4 to 12 Vrms (at approximately 32 Hz) for each opaque electrode (relative to the common back-plane electrode) with a dc component of less than 50mV. The recommended method of providing this from an lcd-driver (having a single 5 volt supply) is to drive the back-plane as well as the electrodes. Opaque segments are driven by a pulsed signal in anti-phase with the signal to the backplane. Electrodes fed by a signal in-phase with the back-plane signal will remain transparent. The recommended waveforms are shown in the Figure below. Communication between the FPGA and the LCD is thus by "coded" binary signals which will drive each of the LCD segments and the LCD backplane. Unused segments cannot be left 'floating' - they should be driven by a signal in-phase with the backplane. A total of 30 "coded" binary signals are thus required to interface the LCD and the FPGA; one for each segment.




The first level in the design hierarchy is thus:

Example VHDL descriptions for each of these components are given below. These examples make use of specific signal sub-types and functions to convert between signal values used internally and those required by the interface specification. These functions and sub-types are defined in a package ('defns') which is also shown below:

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.ALL;

PACKAGE defns IS

 TYPE char_array IS ARRAY (1 TO 8) OF CHARACTER;
 SUBTYPE lcd_seg IS qsim_state_vector(6 DOWNTO 0);
 SUBTYPE lcd_int IS integer RANGE 0 TO 9;

 FUNCTION int_to_seg(number : IN lcd_int) RETURN lcd_seg;

 FUNCTION seg_to_char(segs : IN lcd_seg) RETURN character;
 FUNCTION dp_to_char(dp: IN qsim_state) RETURN character;
 FUNCTION dpcol_to_char(dp, col : IN qsim_state) RETURN character;
 FUNCTION special_to_char(arrow,colon,minus : IN qsim_state) RETURN character;
 FUNCTION k_to_char(k : IN qsim_state) RETURN character;

END defns;


PACKAGE BODY defns IS

 FUNCTION int_to_seg(number : IN lcd_int) RETURN lcd_seg IS
 BEGIN
  CASE number IS
   WHEN 0 => RETURN "1111110";
   WHEN 1 => RETURN "0110000";
   WHEN 2 => RETURN "1101101";
   WHEN 3 => RETURN "1111001";
   WHEN 4 => RETURN "0110011";
   WHEN 5 => RETURN "1011011";
   WHEN 6 => RETURN "1011111";
   WHEN 7 => RETURN "1110000";
   WHEN 8 => RETURN "1111111";
   WHEN 9 => RETURN "1111011";
  END CASE;
 END int_to_seg;

 FUNCTION seg_to_char(segs : IN lcd_seg) RETURN character IS
 BEGIN
  CASE segs IS
   WHEN "1111110" => RETURN '0';
   WHEN "0110000" => RETURN '1';
   WHEN "1101101" => RETURN '2';
   WHEN "1111001" => RETURN '3';
   WHEN "0110011" => RETURN '4';
   WHEN "1011011" => RETURN '5';
   WHEN "1011111" => RETURN '6';
   WHEN "1110000" => RETURN '7';
   WHEN "1111111" => RETURN '8';
   WHEN "1111011" => RETURN '9';
   WHEN "1000111" => RETURN 'F';
   WHEN "0001110" => RETURN 'L';
   WHEN "0111110" => RETURN 'U';
   WHEN "1110111" => RETURN 'R';
   WHEN "1001111" => RETURN 'E';
   WHEN OTHERS => RETURN '?';
  END CASE;
 END seg_to_char;

 FUNCTION dp_to_char(dp: IN qsim_state) RETURN character IS
 BEGIN
  CASE dp IS
   WHEN '1' => RETURN '.';
   WHEN '0' => RETURN ' ';
   WHEN OTHERS => RETURN '?';
  END CASE;
```

```
END dp_to_char;

FUNCTION dpcol_to_char(dp, col : IN qsim_state) RETURN character IS
VARIABLE dpcol : qsim_state_vector(0 TO 1);
BEGIN
 dpcol(0) := dp;
 dpcol(1) := col;
 CASE (dpcol) IS
  WHEN "10" => RETURN '.';
  WHEN "00" => RETURN ' ';
  WHEN "11" => RETURN '!';
  WHEN "01" => RETURN ':';
  WHEN OTHERS => RETURN '?';
 END CASE;
END dpcol_to_char;


FUNCTION special_to_char(arrow,colon,minus : IN qsim_state) RETURN character
IS
 VARIABLE special : qsim_state_vector(0 TO 2);
 BEGIN
  special(0) := arrow;
  special(1) := colon;
  special(2) := minus;
  CASE (special) IS
   WHEN "000" => RETURN ' ';
   WHEN "001" => RETURN '-';
   WHEN "010" => RETURN ':';
   WHEN "011" => RETURN '*';
   WHEN "100" => RETURN '^';
   WHEN "101" => RETURN '=';
   WHEN "110" => RETURN 'T';
   WHEN "111" => RETURN '#';
   WHEN OTHERS => RETURN '?';
  END CASE;
 END special_to_char;

 FUNCTION k_to_char(k : IN qsim_state) RETURN character IS
 BEGIN
  CASE k IS
   WHEN '1' => RETURN '1';
   WHEN '0' => RETURN ' ';
   WHEN OTHERS => RETURN '?';
  END CASE;
 END k_to_char;

END defns;
```

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.ALL;
USE WORK.defns.ALL;


ENTITY lcd IS
 PORT(dig1, dig2, dig3 : IN lcd_seg;
   k, p1, p2, p3, c1: IN qsim_state;
   arrow, colon, minus, bp : IN qsim_state;

   display : OUT char_array);
END lcd;

ARCHITECTURE behav OF lcd IS
BEGIN
 internal_action: PROCESS(dig1,dig2,dig3,k,p1,p2,p3,c1,arrow,colon,minus,bp)

 VARIABLE dig1_bp,dig2_bp,dig3_bp : lcd_seg;
```

```
VARIABLE k_bp,p1_bp,p2_bp,p3_bp,c1_bp : qsim_state;
VARIABLE arrow_bp,colon_bp, minus_bp : qsim_state;

BEGIN
 FOR seg IN 0 TO 6 LOOP
  dig1_bp(seg) := dig1(seg) XOR bp;
  dig2_bp(seg) := dig2(seg) XOR bp;
  dig3_bp(seg) := dig3(seg) XOR bp;
 END LOOP;

 k_bp := k XOR bp;
 p1_bp := p1 XOR bp;
 p2_bp := p2 XOR bp;
 p3_bp := p3 XOR bp;
 c1_bp := c1 XOR bp;
 arrow_bp := arrow XOR bp;
 minus_bp := minus XOR bp;
 colon_bp := colon XOR bp;


 display(1) <= special_to_char(arrow_bp,colon_bp,minus_bp);
 display(2) <= k_to_char(k_bp);
 display(3) <= dp_to_char(p1_bp);
 display(4) <= seg_to_char(dig1_bp);
 display(5) <= dpcol_to_char(p2_bp,c1_bp);
 display(6) <= seg_to_char(dig2_bp);
 display(7) <= dp_to_char(p3_bp);
 display(8) <= seg_to_char(dig3_bp);


 END PROCESS;

END behav;
```

```
ARCHITECTURE behav OF fpga IS
                    -- BEHAVIOURAL DESCRIPTION FOR fo = 10kHz
 SIGNAL count, freq : integer := 0;
                     -- SIGNALS TO ALLOW MONITORING OF INTERNAL ACTION
SIGNAL bp_state : qsim_state := '0';
BEGIN

 int_action: PROCESS(fo, fu, reset, ppm, freq, count)
                         -- CALCULATE fu FREQUENCY FROM PERIOD

 BEGIN
  IF reset='1' THEN
   count <= 0;
   freq <= 0;
  ELSE
   IF fo'last_value='0' AND fo'event THEN
    count <= count + 1;
   END IF;
   IF fu'last_value='0' AND fu'event THEN
    IF count /= 0 THEN
     IF ppm='0' THEN freq <= 1000000/count;  -- CALCULATE fu IN HZ
                                             -- CALCULATE fu IN PPM
     ELSE freq <= 600000/count;
     END IF;
    ELSE freq <= 0;
    END IF;
    count <= 0;
   END IF;
  END IF;
 END PROCESS int_action;

 generate_backplane: PROCESS(fo, bp_state)
                                          -- GENERATE LCD BACKPLANE
  SIGNAL VARIABLE fo_count : integer := 0; BEGIN
```

```
 IF fo'event AND fo'last_value='1'
  THEN fo_count:= fo_count + 1;
 END IF;
 IF fo_count = 100
  THEN bp_state <= NOT(bp_state);
                          -- TOGGLE bp SIGNAL TO GIVE FREQ OF 50HZ
  (fo=10kHz) fo_count := 0;
 END IF;
 bp <= bp_state;
END PROCESS generate_backplane;


generate_7seg_outputs: PROCESS(bp_state)
VARIABLE temp_val1, temp_val2, temp_val3 : integer;
VARIABLE temp_lcd1, temp_lcd2, temp_lcd3 : lcd_seg;
BEGIN

 -- NB THIS PROCESS IGNORES OVERFLOW & UNDERFLOW IN VALUE OF freq

 temp_val3 := freq REM 10;
 temp_lcd3 := int_to_seg(temp_val3);
 FOR seg IN 0 TO 6 LOOP
  dig3(seg) <= temp_lcd3(seg) XOR bp_state;
 END LOOP;

 temp_val2 := (freq/10) REM 10;
 temp_lcd2 := int_to_seg(temp_val2);
 FOR seg IN 0 TO 6 LOOP
  dig2(seg) <= temp_lcd2(seg) XOR bp_state;
 END LOOP;

 temp_val1 := (freq/100) REM 10;
 temp_lcd1 := int_to_seg(temp_val1);
 FOR seg IN 0 TO 6 LOOP
  dig1(seg) <= temp_lcd1(seg) XOR bp_state;
 END LOOP;
END PROCESS generate_7seg_outputs;

p1 <= bp_state;
     -- GENERATE SIGNALS FOR OTHER SEGMENTS
p3 <= bp_state;
p2 <= NOT(bp_state) WHEN ppm='1' ELSE bp_state;
k <= bp_state;
c1 <= bp_state;
arrow <= bp_state;
colon <= bp_state;
minus <= bp_state;

END behav;
```

The FPGA description is an example only and is incomplete in several respects. Several design aspects have been left for you to consider, eg:

1. Is the choice of fo = 10kHz sufficient or excessive? Values too low will degrade the resolution, values too high will raise the FPGA power dissipation unnecessarily.
2. Should the decimal point (p2) be turned on as soon as the ppm_hz signal goes active (as in the example above), or should it stay off until the result (in ppm) is known? (And vice versa.)
3. How will the LCD show under and over-range conditions (ie fu<0.50 Hz and fu>5.00Hz)? Note that unused LCD segments should not be left 'floating'.

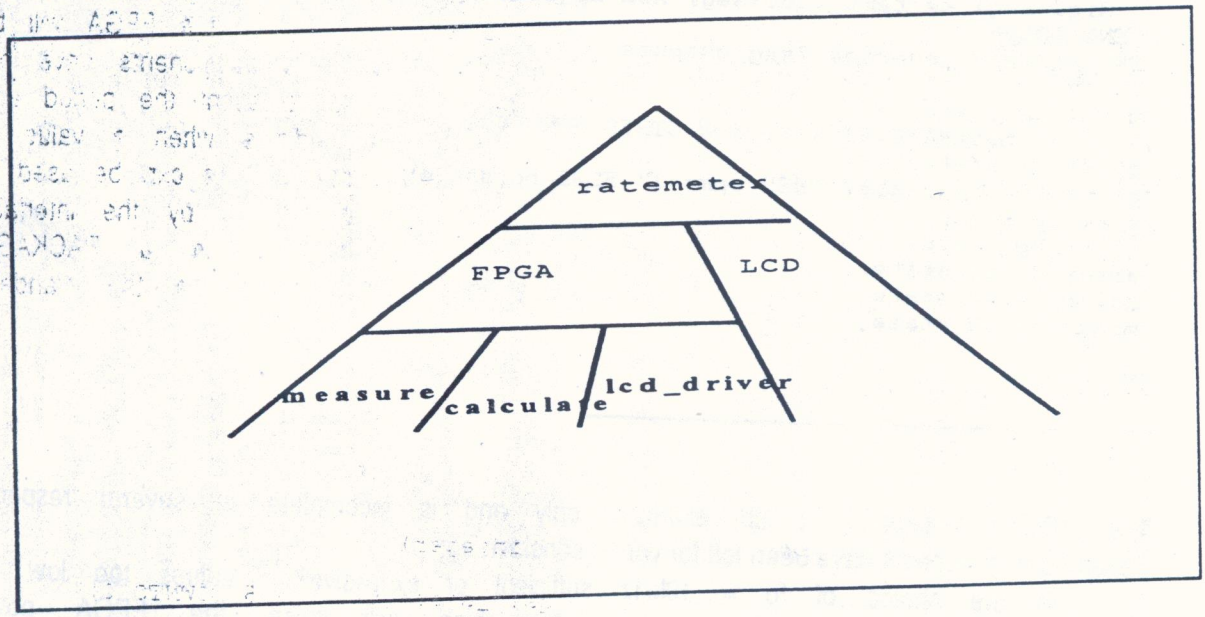
If you do not make use of a segment, drive it with a signal in-phase with the back-plane (bp).

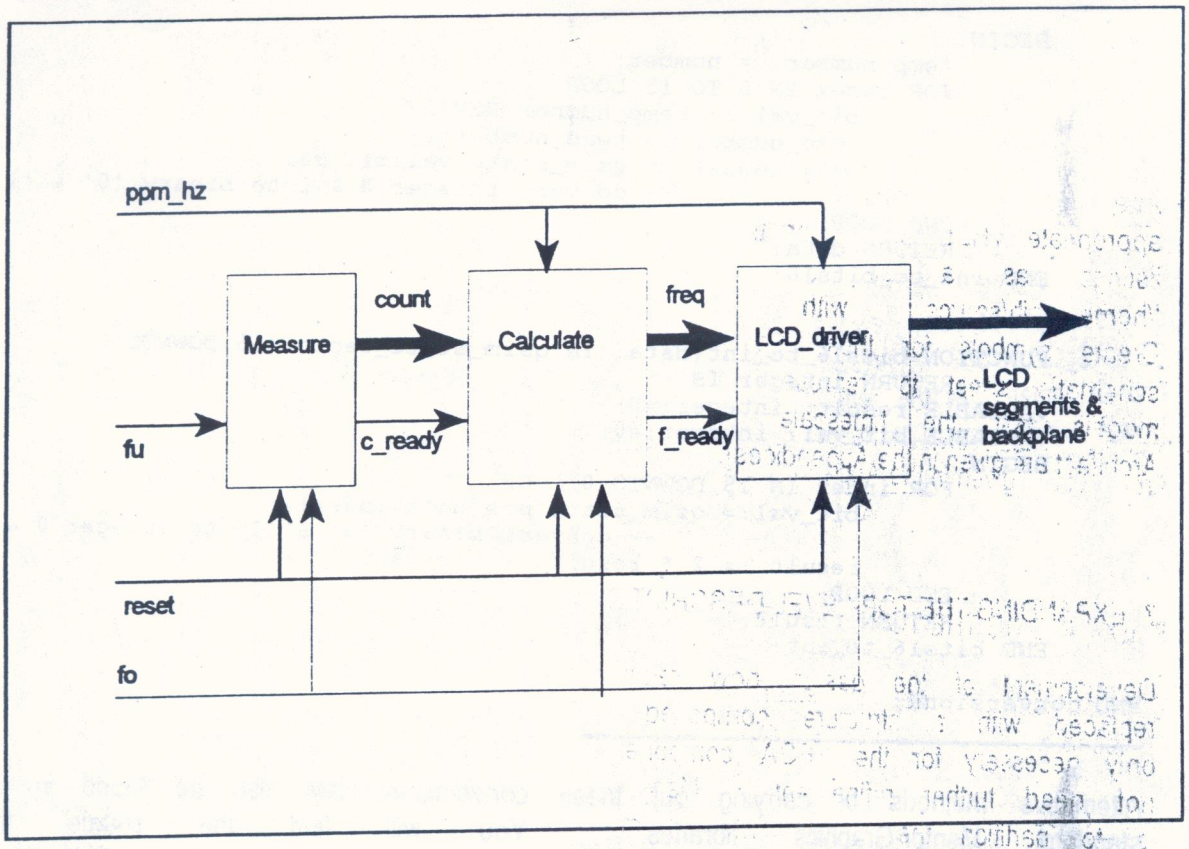
4. Should leading-zeros be displayed? No

After you have considered these points, use the Design Architect tool to create appropriate VHDL models for the FPGA and the LCD. You can use the above model listings as a starting point; copies can be found in the directory '/homedir/sjh/source' with filenames 'defns.txt', 'lcd.txt' and 'fpga.txt'. Create symbols for these components, then place and connect the symbols on a schematic sheet to represent the ratemeter. Simulate the system to verify your models (set the timescale units to seconds). [Instructions for using Design Architect are given in the Appendices].

## 7. EXPANDING THE TOP-LEVEL DESCRIPTION

Development of the design now requires the top-level behavioural description to be replaced with a structure composed of lower-level behavioral functions. This is only necessary for the FPGA component - the behavioural LCD model definition does not need further refinement. Many possible architectures exist, and one solution is to partition the design into a measure stage (to measure the period of the unknown clock), a calculate stage to calculate the required unknown frequency, and a driver stage to drive the LCD with the necessary "coded" signals.






As before, the interface specification for each of these components needs to be defined before the model is created. The inputs and outputs from the FPGA will be as defined at the top-level, however the signals between the components have not been defined. These will be binary, vectors being used to represent the period and frequency values of fu, and simple flags being used to indicate when a value is ready for the next component. As with the FPGA component, functions can be used to convert between internal signal types and the binary type required by the interface specifiation. Possible examples using 16-bit data, for use in a PACKAGE ('conversions'), are given below. Note that these functions assume the standard Mentor-Graphics convention of MSB-first (left-most).

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.ALL;

PACKAGE conversions IS
      FUNCTION int_to_bits16(number: IN integer)
            RETURN qsim_state_vector;
      FUNCTION bits16_to_int(data: IN qsim_state_vector(15 DOWNTO 0))
            RETURN integer;

END conversions;

PACKAGE BODY conversions IS

      FUNCTION int_to_bits16(number: IN integer)
            RETURN qsim_state_vector IS
      VARIABLE data : qsim_state_vector(15 DOWNTO 0) := "0000000000000000";
      VARIABLE bit_val, temp_number : integer:= 0;
```

```
    BEGIN
        temp_number := number;
        FOR index IN 0 TO 15 LOOP
            bit_val := temp_number REM 2;
            temp_number := temp_number/2;
            data(index) := qsim_state'val(bit_val);
                -- convert integer 0 & 1 to binary '0' & '1'
        END LOOP;
        RETURN data;
    END int_to_bits16;

    FUNCTION bits16_to_int(data: IN qsim_state_vector(15 DOWNTO 0))
        RETURN integer IS
    VARIABLE result: integer:=0;
    VARIABLE bit_val: integer:=0;
    BEGIN
        FOR index IN 15 DOWNTO 0 LOOP
            bit_val:= qsim_state'pos(data(index));
                -- convert binary '0' & '1' to integer 0 & 1
            result := 2 * result  + bit_val;
        END LOOP;
        RETURN result;
    END bits16_to_int;

END conversions;
```

Alternative methods of carrying out these conversions may also be found in the standard Mentor-Graphics libraries. You will find the source files ('qsim_logic_header.hdl' and 'qsim_logic_body.hdl') in directory '/idea.new/pkgs/sys_1076_std.any/src'.

Again, there are several design issues which you are left to consider. In particular:

1. How many bits are needed to represent the period and frequency values?
2. What will the outputs be on reset, and in the under/over-range conditions?

Write suitable VHDL behavioural models for each of these components, create symbols, and then place on a schematic sheet for the FPGA component. Change the MODEL property on this symbol to reference the schematic instead of the initial behavioural model, then resimulate the Ratemeter to validate your new models (you can use the same test data). [Refer to the Appendices for detailed instructions]

## 8. THE REGISTER TRANSFER LEVEL (RTL)

The next step in the design process is to convert the behavioural MEASURE, CALCULATE, and LCD_DRIVER building blocks into structures composed of even simpler behavioural sections. Possible solutions are outlined in the following sections. Because the behaviour of these blocks is now chiefly concerned with the movement of data between various latches or registers, this level of description is commonly

called the Register Transfer Level (RTL). The movement of data is controlled according to a flow-chart description, which can be implemented by a Finite State Machine (FSM).

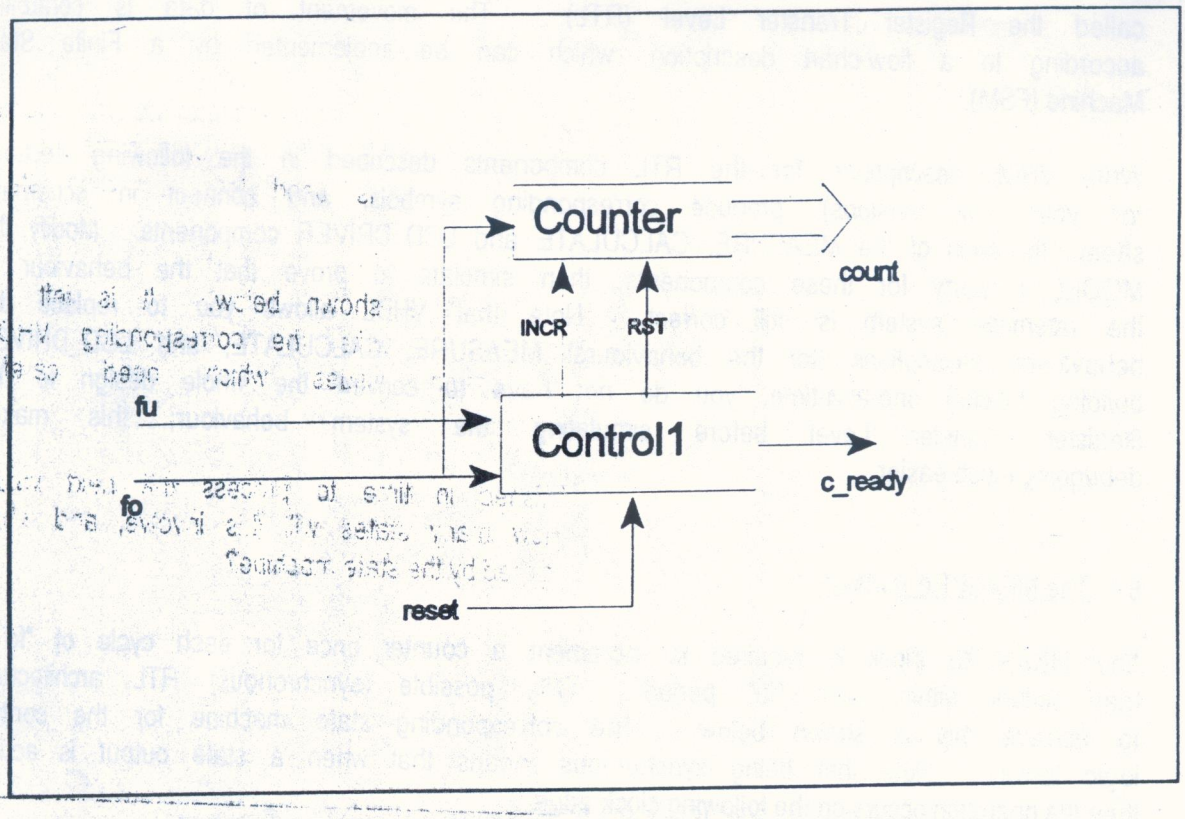
Write VHDL descriptions for the RTL components described in the following sections (or your own versions), produce corresponding symbols, and connect in schematic sheets for each of the MEASURE, CALCULATE and LCD_DRIVER components. Modify the MODEL property for these components, then simulate to prove that the behaviour of the ratemeter system is still correct. Note that VHDL allows you to replace the behavioural descriptions (for the behavioural MEASURE, CALCULATE, and LCD_DRIVER building blocks) one-at-a-time, you do not have to convert the whole design to the Register Transfer Level before simulating the system behaviour; this makes debugging much easier.

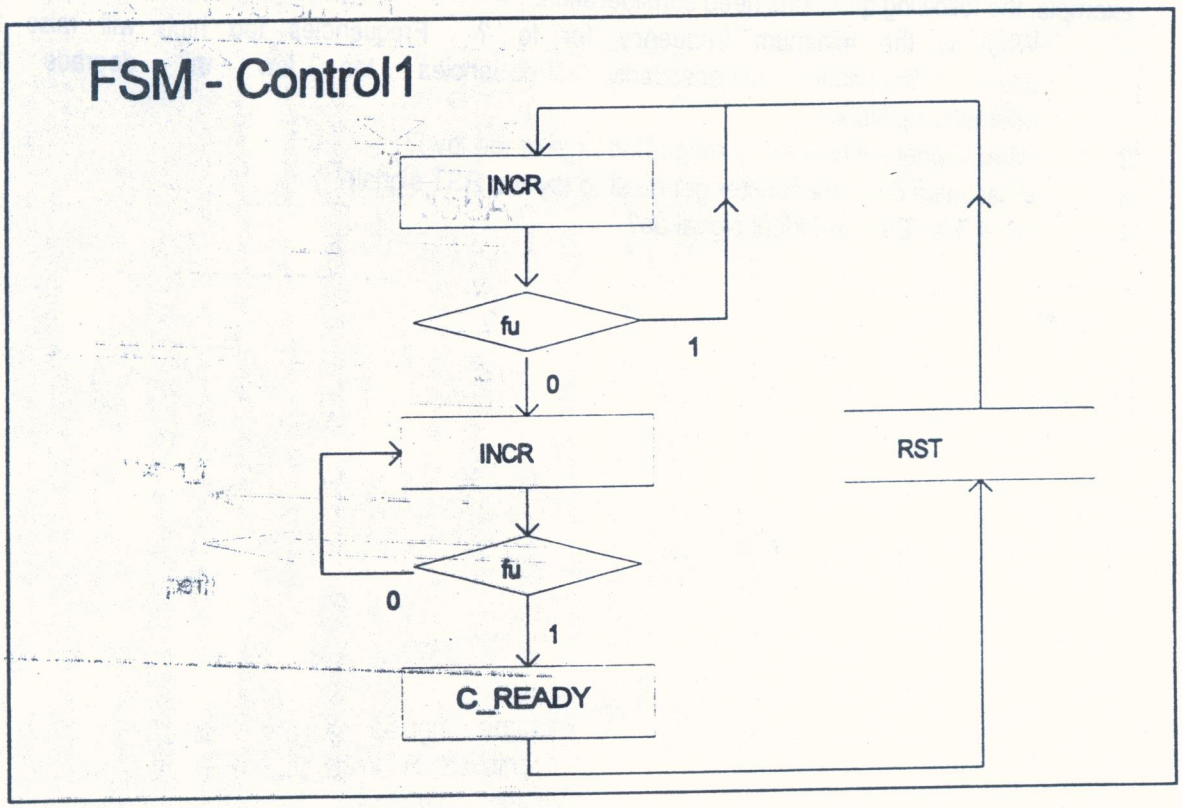
## 8.1. The MEASURE partition

The MEASURE block is required to increment a counter once for each cycle of "fo" that occurs within each "fu" period. One possible (synchronous) RTL architecture to achieve this is shown below. The corresponding state machine for the control logic follows. Note that being synchronous means that when a state output is active then the operation occurs on the following clock edge.

Several design issues are incompletely specified in these descriptions. For example, the following questions need consideration:

1. What is the minimum frequency for fo ? Frequencies too high will raise the power dissipation unnecessarily, frequencies too low will degrade the ratemeter resolution.
2. What happens if fu is out of range (too high or too low)?
3. What value does the counter get reset to (by the RST signal)?
4. What does the reset input signal do?


Counter
count
INCR
RST
fu
Control1
c_ready
fo
reset

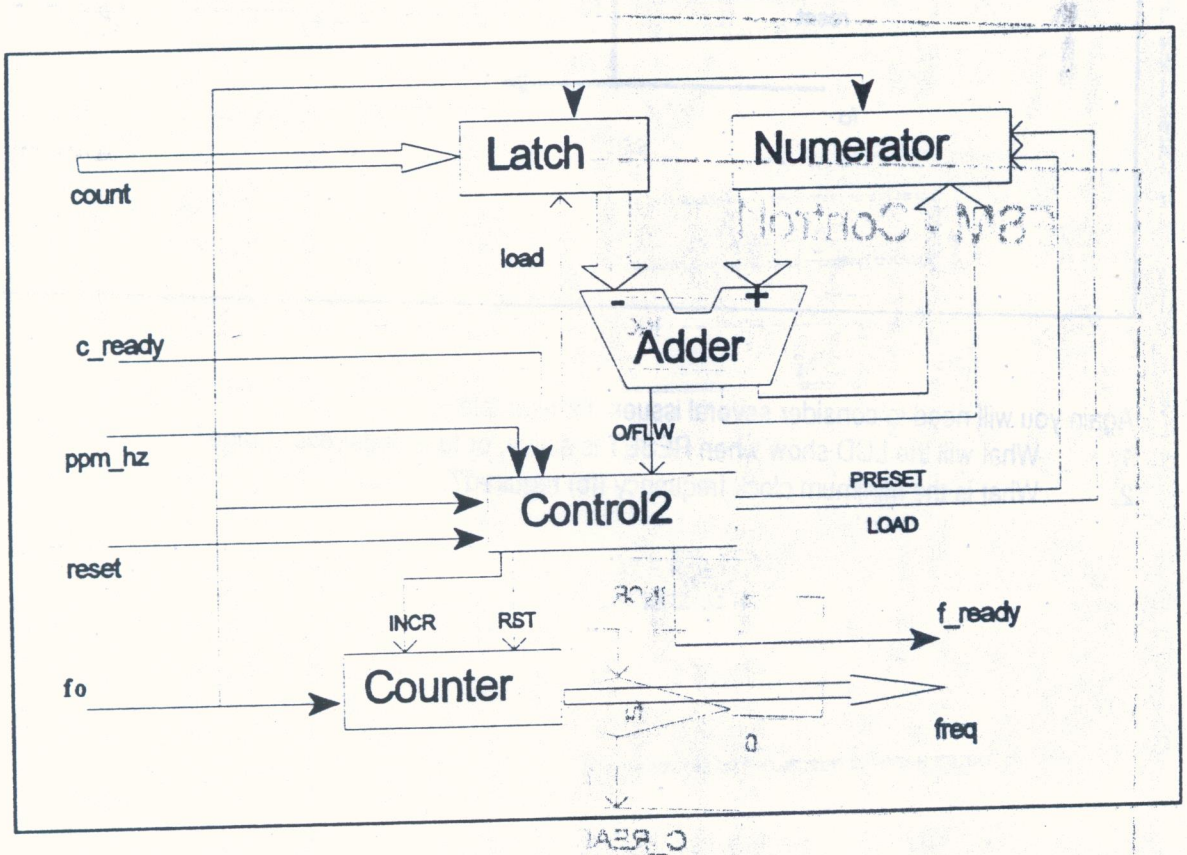

FSM - Control1
INCR
fu
1
0
INCR
RST
fu
0
1
C_READY

## 8.2. The CALCULATE partition

Division essentially relies upon repeated subtraction. and the simplest method of implementing the CALCULATE block is to use a state machine which continually subtracts the COUNT value from the numerator until the solution is zero or negative. The number of subtraction operations which is used to reach this state is the required frequency value.

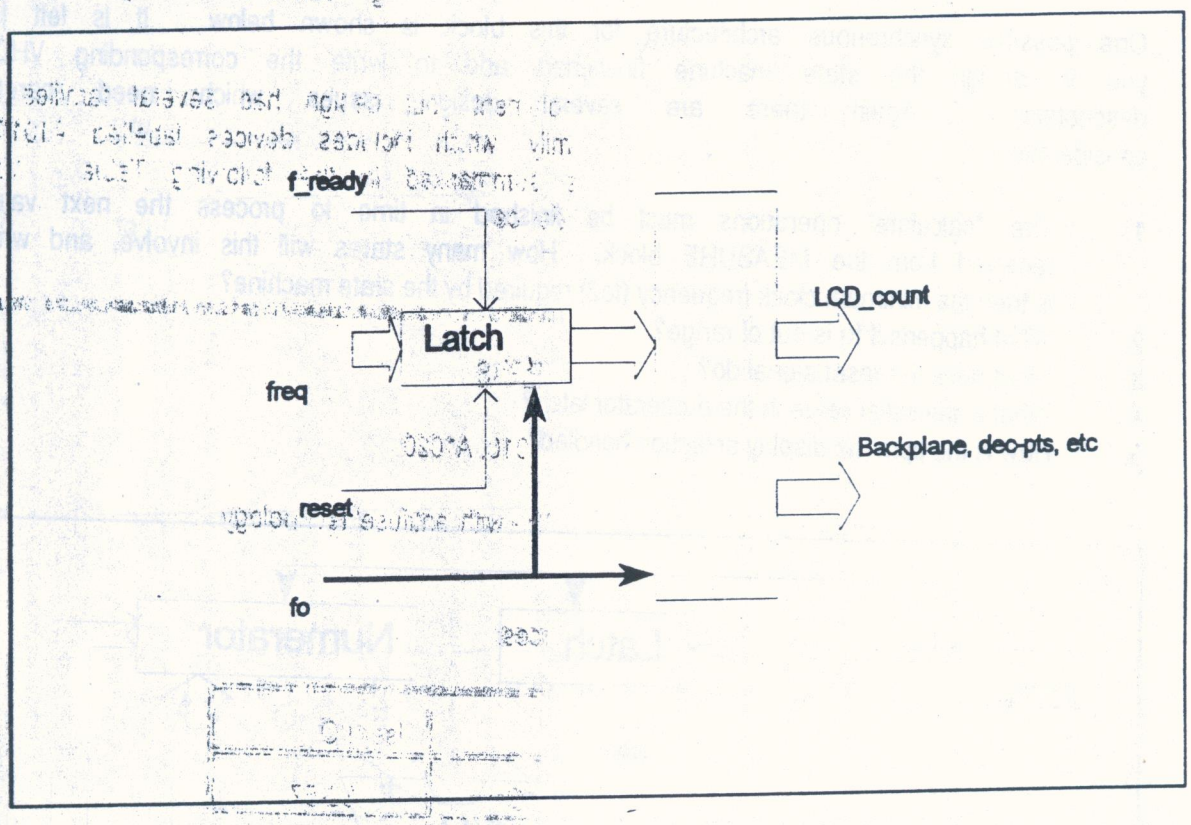
One possible synchronous architecture for this block is shown below. It is left for you to design the state machine flow-chart and to write the corresponding VHDL descriptions. Again there are several design issues which need careful consideration:

1. The "calculate" operations must be finished in time to process the next value received from the MEASURE block. How many states will this involve, and what is then the minimum clock frequency (fo2) required by the state machine?
2. What happens if fu is out of range?
3. What does the reset signal do?
4. What is the initial value in the numerator latch?
5. How is the ppm_hz display selection handled?



## 8.3. The LCD Driver partition

The LCD_driver is required to send results to a 3.5 digit LCD. At the RT-level of description this partition can be implemented by a simple latch which stores the result of each successive "inversion" operation carried out by the CALCULATE block, and a coder to format the result into the form required by the LCD. The figure below shows a possible synchronous architecture.




Again you will need to consider several issues, for example:

1. What will the LCD show when RESET is active, or fu is under/over-range?
2. What is the minimum clock frequency (fo) required?

## 9. THE GATE-LEVEL DESCRIPTION

At this point you should now have an RTL level description of the system which meets the specification fully. Hopefully the main advantage of following a top-down design methodology is becoming clear: you have been able to ensure that the system algorithms and architecture are correct before you have to worry about how you will actually implement it. The next step in the design process is to transform your VHDL RTL descriptions into a gate-level circuit using conventional digital design techniques.

The Actel FPGA which you will use to implement your design has several families of devices. You will be using the ACT1 family, which includes devices labelled A1010A and A1020. The properties of these are summarised in the following Table. You should aim to fit your design onto the larger A1020 device.

ACTEL FPGAS

Act1 Devices: A1010, A1020

2µm Si-gate 2-level metal CMOS - with 'antifuse' technology

Resources:

| Device | Modules | Gates | User I/O |
|---|---|---|---|
| A1010 | 295 | 1200 | 34-57 |
| A1020 | 547 | 2000 | 34-69 |

Vcc = -0.5 to +7 volts
Vi and Vo = -0.5 to (Vcc+0.5) volts

Typical module delay = 5 to 15 ns
('long track' delay = 15 to 35 ns)

Maximum clock rate up to 100MHz

Although the Table lists a gate-count figure for each array-size, it should be emphasised that this is only an approximate figure. The actual gate count achievable depends on the type of gate which is being used. The Actel cells are based upon multiplexer modules which can be programmably-configured to act as

various types of gate, and thus a 2-input NAND gate will have the same module-count as a more complex 3-input XOR-AND gate. To make most efficient use of an Actel array you should design using multiplexers rather than more basic NAND/NOR structures. It is also recommended that FSMs are implemented using one register-per-state so as to minimise the amount of decoding logic required, rather than trying to minimise the number of state registers. The attached Actel Applications Notes give various guidelines for efficient design procedures. Don't forget that CMOS gate arrays are particularly prone to logic 'hazards' , where unexpected 'glitches' may occur due to differences in propagation delay along different paths. DON'T 'gate' signals to the clock input of registers.

The Actel library includes several high-level functions (such as counters) which should be used whenever possible. These components have been carefully designed to minimise the required module-count and to maximise ther performance. Two types of component exist: hard-macros which have a fixed internal wiring and hence a fixed propagation delay, and soft-macros which are wired as appropriate and hence have a delay which is dependent upon the place-and-route stage. A hard-macro is a fixed entity and cannot be changed, however a soft-macro can be copied and then edited if its functionality does not quite meet your requirements. Note that in this case unused outputs are allowable, but unused inputs are NOT.

Real electronic components have a limited capacity for driving capacitive and resistive loads, and these parasitics can add delay to signal transitions and increase the rise and fall-times. One area where high loading is particularly likely to be a problem is on the clock line used to drive the system flip-flops (as present in the FSMs). The clock must be distributed in such a way as to minimise the load on the clock driver, but also so as to maintain synchronism in the system. The Actel FPGAs overcome this problem by providing a special "CLOCK" input buffer which is able to distribute a clock signal over the entire array, however any other signals which require distributing to a large number of components (ie with a high fanout) cannot make use of this facility. It is recommended that you adopt a "tree" structure for distributing such signals, using equal numbers of buffers in each branch of the tree. As a guideline you should try to restrict the fanout of any signal to less than 10, with an absolute upper limit of 20 (the place-and-route software will not work if extremely high fanouts are used).