Chapter 2

68HC11 Based Temperature Monitoring Board

Before we discuss the testing approach, it is helpful to know the system on which it is being applied. In this chapter, we will describe the design and operation of the system and the way in which it is modeled in VHDL.

2.1 Circuit Operation

The function of the system is temperature monitoring. It is built around the Motorola 68HC11A8 microcontroller. A simplified schematic of the system is shown in Figure 2.1. The system program is stored in the external EEPROM. The internal RAM of the microcontroller is used during the program execution. A latch is used for address/data demultiplexing, whereas a decoder is used for address decoding. The purpose of external RAM will be explained in Chapter 4. A temperature transducer senses the temperature and outputs the corresponding analog output voltage. The transducer is so calibrated that +5 V analog output represents +127 ^{0}C whereas 0 V output represents –128 ^{0}C. The operation of the transducer is assumed to be linear. This analog voltage is applied to an analog to digital converter (ADC) which outputs the eight bit unsigned number representing the temperature. Hence, corresponding to 5 V, it sends out 1111 1111 , whereas for 0 V, it outputs 0000 0000.

The ADC output is applied to one port of the Programmable Peripheral Interface (PPI) through which the microcontroller reads this value. Microcontroller XORs this value with 1000 0000.

This operation makes the value of the signed data equal to the actual temperature. All these steps are shown graphically in Figure 2.2

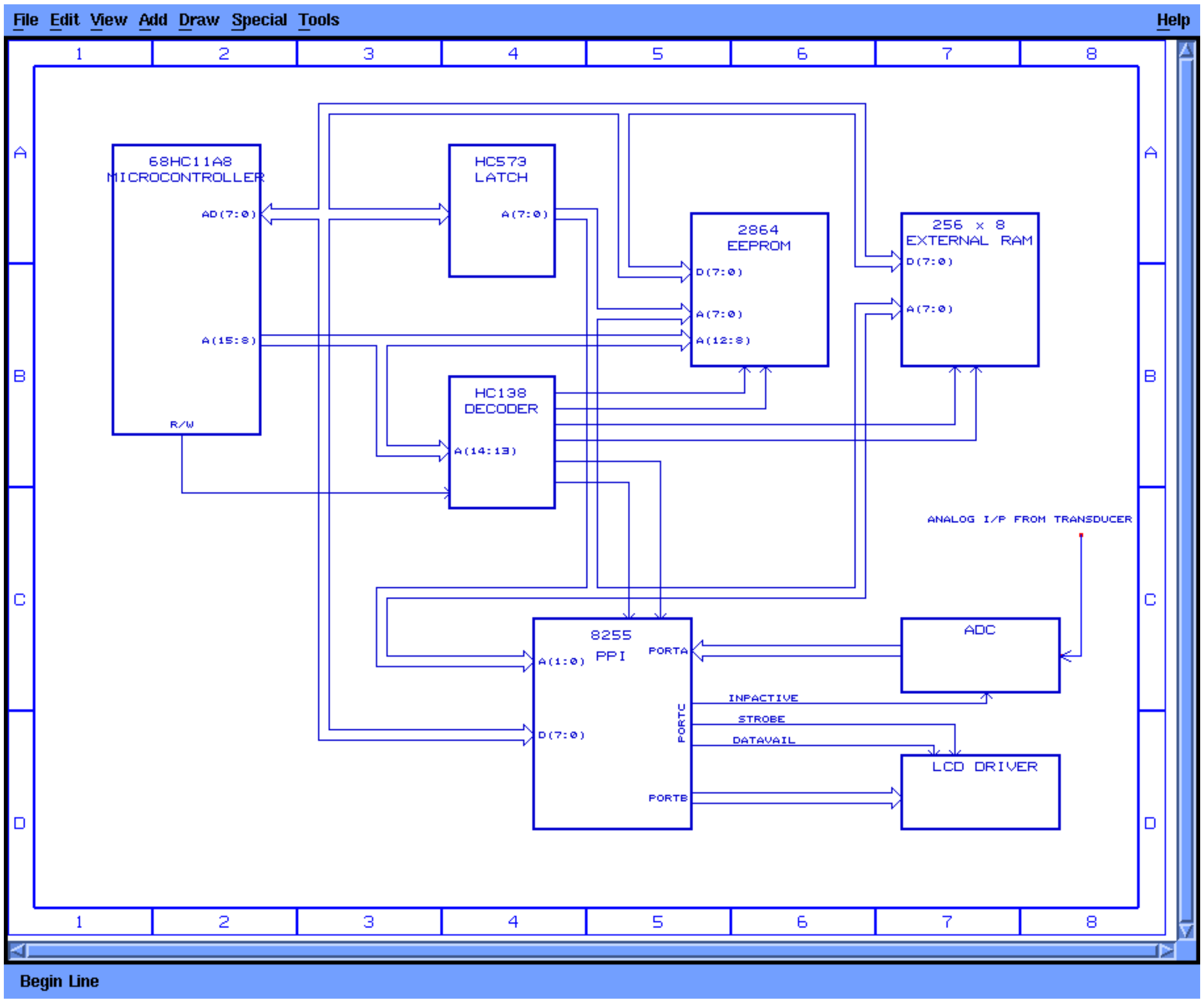


Figure 2.1 Schematic diagram for temperature controller

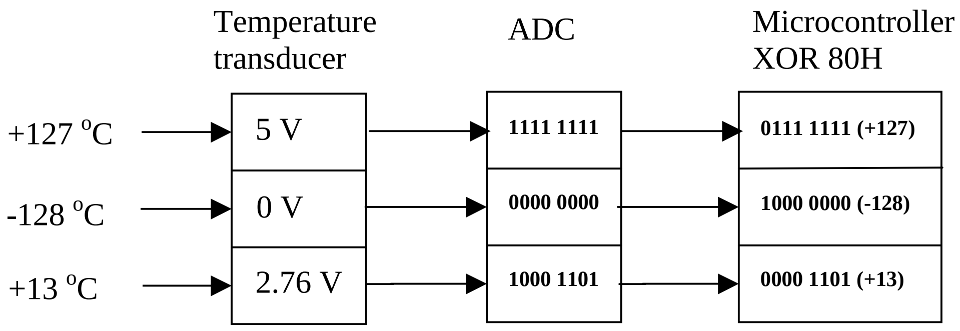


Figure 2.2 Data formats for temperature controller

Next, the microcontroller computes the ASCII codes for each numeral that makes up the temperature value, and sends the serial stream to the LCD driver through the output port of PPI. The algorithm for converting a signed binary number to a series of ASCII characters representing the decimal value of the number is as follows:

1. Check the MSB of the number to determine whether it is positive or negative
2. If it is positive, send ASCII '+' to output, store the number in the accumulator and go to step (4)
3. If it is negative, send ASCII '-' to output, take two's complement of the number, store it in accumulator and go to step (4)
4. Check the number in the accumulator.

    If number ≥ 120 D, add 168 D to it and go to step (5), else

    If number ≥ 110 D, add 162 D to it and go to step (5), else

    If number ≥ 100 D, add 156 D to it and go to step (5), else

    If number ≥ 90 D, add 54 D to it and go to step (5), else

    If number ≥ 80 D, add 48 D to it and go to step (5), else

    If number ≥ 70 D, add 42 D to it and go to step (5), else

If number ≥ 60 D, add 36 D to it and go to step (5), else

If number ≥ 50 D, add 30 D to it and go to step (5), else

If number ≥ 40 D, add 24 D to it and go to step (5), else

If number ≥ 30 D, add 18 D to it and go to step (5), else

If number ≥ 20 D, add 12 D to it and go to step (5), else

If number ≥ 10 D, add 6 D to it and go to step (5), else

go to step (5)

5. Check the carry, if it is clear, go to step (6) , if it is set, it means the result is greater than 100 D. So output the most significant 1 to the output by sending 31 H (ASCII ‘1’)
6. The data in accumulator now contains packed BCD result. So, unpack the two digits, convert them to ASCII by ORing with 30 H and then send them one by one to the output.

The ASCII conversion process is shown in Figure 2.3

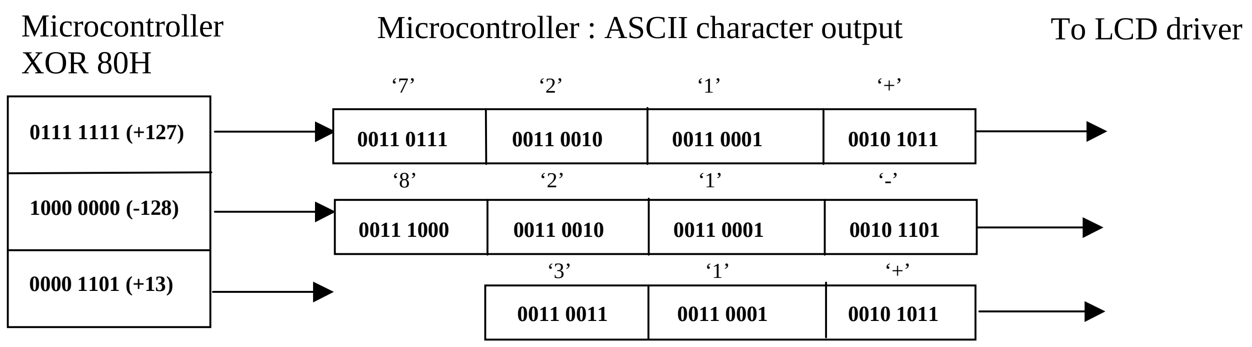


Figure 2.3 ASCII conversion process for temperature controller

2.2 Hardware description of the circuit

In the first phase of the project, the system was modeled in VHDL and self-test was run from the model. In the second phase, the code was run from an actual system. There is a slight difference between the VHDL model and the actual system. This difference lies mostly in the operation of the LCD driver which is used for displaying system's normal operation and test results messages. We will describe below the VHDL model of the system as shown in Figure 2.1. The implemented system is assumed to be similar unless the difference is stated.

The initial work on VHDL modeling of the system was done by Jason Liu. I made the models of the LCD driver and the ADC. Later on, I also added an external RAM and associated circuitry to the system. The components of the system are briefly described below:

The *68HC11A8 microcontroller* has 64 K of addressable space. It has 256 bytes of internal RAM and 8 Kbytes of internal ROM. It has five ports A,B,C,D and E. The ports B and C become the address/data lines when the microcontroller is used in the expanded mode. In the expanded mode, external chips are connected to the microcontroller to make the whole system. In this project, the microcontroller is used in the expanded mode. The microcontroller used in the actual system was 68HC11E9 which has the same pins and ports as the 68HC11A8 but has 512 bytes of internal RAM.

The *crystal* used in the circuit runs at 4 MHz. The microcontroller operates at ¼th of the crystal frequency i.e. at 1 MHz.

An external *8 KB X2864A EEPROM* is used to store the program code.

The interrupt vector addresses are located in the internal 8 KB ROM by default. So, the interrupt vectors are placed at the interrupt vector addresses in the internal ROM. Also, after power up the system starts executing instructions from the starting address of the internal ROM. Since we are using the external EEPROM for storing the program, we write the opcode for jump to external EEPROM at the starting location of internal ROM.

The *74HC573 latch* is used to demultiplex addresses from the shared address/data lines.

A *74HC138 decoder* is used for address decoding of external chips.

An *8255 Programmable peripheral interface (PPI)* is used to get additional ports for input-output operations. The PPI is programmed in mode 0 with its port A and upper half of port C acting as input ports, while port B and the lower half of port C act as output ports. The digital output of the ADC is applied to the input port A while the ASCII data representing the temperature value is send to output port B. The lower half of port C is used for sending control signals.

The *analog to digital converter (ADC)* takes an analog input voltage and converts it into an 8-bit digital signal. The analog input voltage ranges from 0 to 5 volts with the corresponding digital output ranging from 0000 0000 to 1111 1111. It accepts the input voltage and does the conversion process only when the control signal INPACTIVE is high. This control signal comes from line 0 of port C of PPI which is made high through the system program. In the system implementation, we applied the digital value directly to the PPI through Dual-inline-Package (DIP) switches. There is no INPACTIVE signal for the implemented system since there is no A/D conversion.

The *LCD driver* takes the 8-bit ASCII code of the characters as input and displays the corresponding characters. There are two control signals connected to the LCD driver. When we want to send a stream of characters, we should tell the LCD driver when this stream starts and when it ends, and secondly we should tell it when the next character is to be send. So, the first control signal DATAVAIL is high as long as the microcontroller is sending the ASCII data to the LCD driver. The LCD driver puts all the ASCII data in its data buffer when the DATAVAIL signal is high and displays the corresponding characters when the DATAVAIL signal goes low. The DATAVAIL signal comes from line 2 of PORT C of PPI. The second control signal STROBE comes from line 1, PORTC of PPI. Before an ASCII data is to be sent, this signal is made high. The LCD driver then treats the data at its input to be the valid ASCII data. Then the STROBE is made low. It is again made high before sending the next character and this process is repeated for all the characters.

For the actual system, we used the Hitachi HD44780 16 character $\times$ 2 line Dot matrix LCD Module. The data lines of the module again connect to PORT B of the PPI. The values sent by the system on this port are treated as either data or instruction values depending on the mode of operation. The mode of operation is selected by the RS pin of the module. Initialization software is also required to be written for the module. Typically, during the initialization, the parameters set are cursor position, cursor blink/no blink, cursor and/or display right/left shift etc. Two control signals were used from the PPI to the module. One was connected to the RS pin to select the data or instruction mode, the other was connected to the enable E input of the module. The module works only when the enable input is set to 1. The initialization software is included in the main program given in Appendix A

2.3 System Modeling in VHDL

We made a chip level structural model of the system in VHDL. For the chips 68HC11A8, X2864A, P8255, 74HC138 and 74HC573, we used the VHDL models from the Synopsys Smartmodel™ library. For the other components i.e. ADC, LCD driver, crystal etc., we developed our own behavioral VHDL models using the Synopsys Simulation Graphical Environment (SGE). All the components were put together in SGE to form the complete system.

This system was then simulated using Synopsys VHDL simulator/debugger to confirm the operation.

2.3.1 Smart model components usage

The step by step operation for using the Smart model library components is given below.

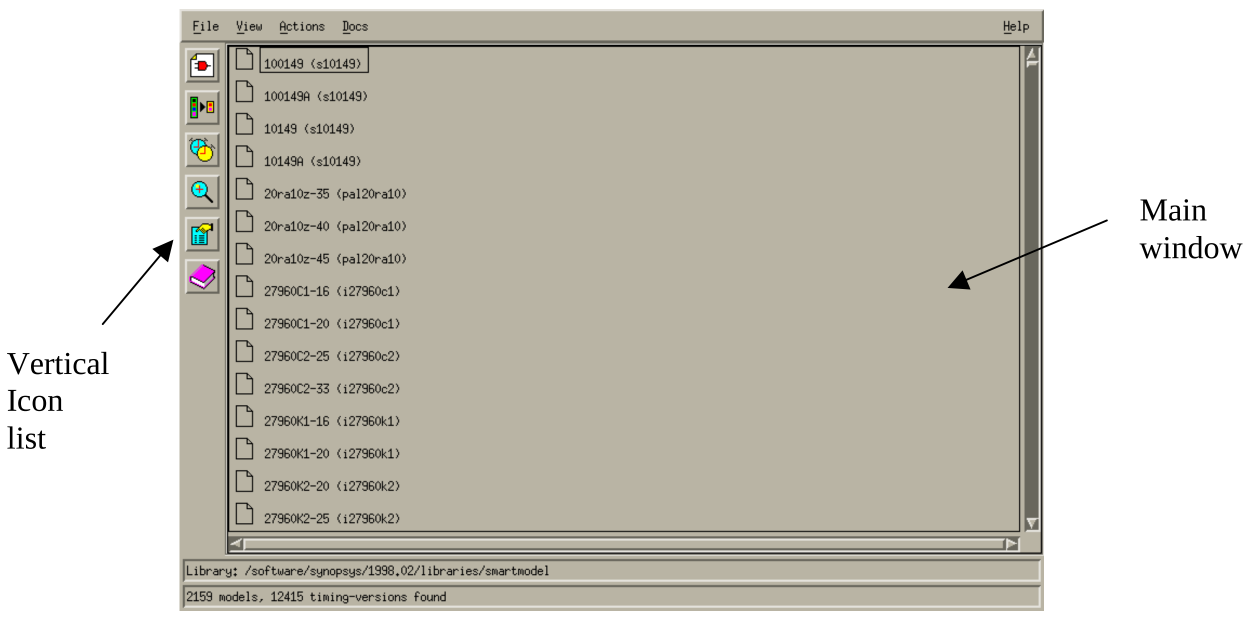
1. Open the Smart model library browser using the command:

```
%  sl_browser
```

The introductory window of the library browser is shown in Figure 2.4. The main window shows all the models of components. Due to the large number of components, it is very difficult to find a component by scrolling this window. To avoid this, we can use the filter dialog box. It is invoked by selecting the filter option in the Actions menu or by clicking on the second item from the top in the vertical icon list on the left side. The filter dialog box is shown in Figure 2.5

There are several filtering options. We can filter by (1) string of characters that appear in the model name, (2) manufacturer's name, (3) function/sub-function, or (4) licensed package name. The results of the selected filtering process appear in the main window.

For example, in Figure 2.5, we are searching for the Intel ROMs which have a 64 in their part name. We put *64 in the string search option, select Intel in the Vendors name list, and select ROM in the function/sub-function list. We keep the licensed package name option disabled since we are not looking for a specific package. After setting these options, we press the Filter button. This starts the filtering process and displays the result in the main window. The results are shown in Figure 2.6. After finding the desired component, we can select the option "Display data sheet" in the Actions menu to find more detailed information.



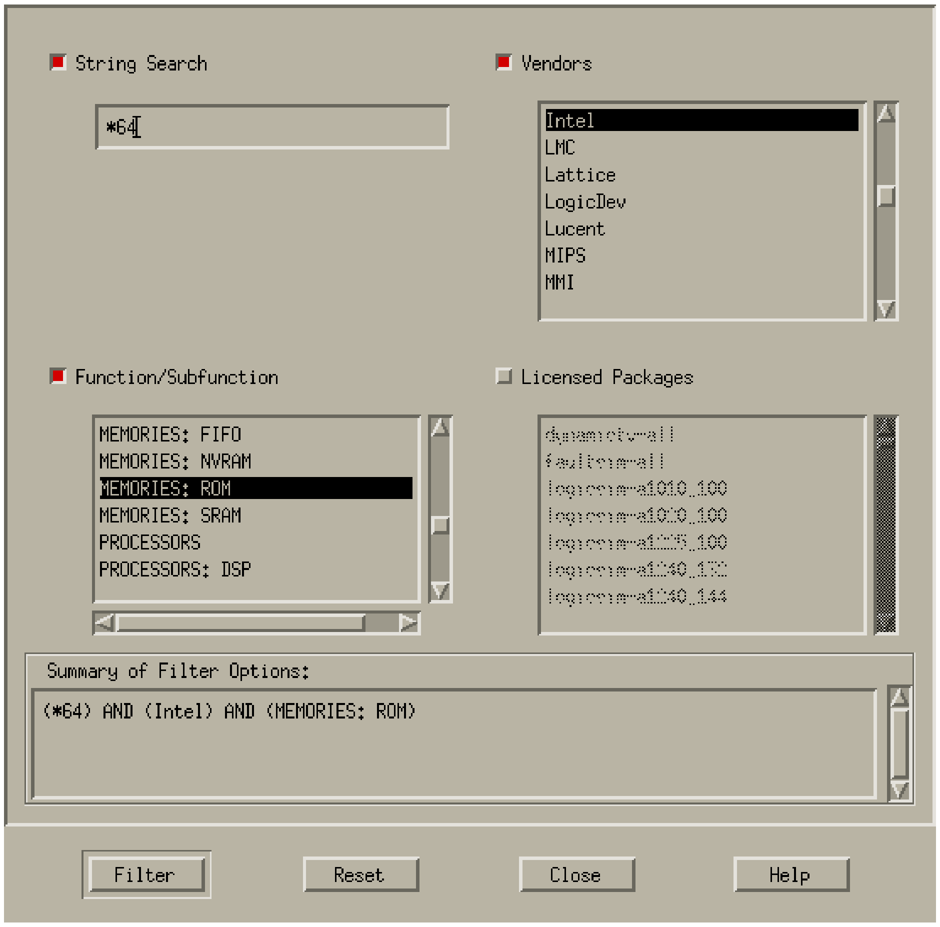
```
100149 (s10149)
100149A (s10149)
10149 (s10149)
10149A (s10149)
20ra10z-35 (pal20ra10)
20ra10z-40 (pal20ra10)
20ra10z-45 (pal20ra10)
27960C1-16 (i27960c1)
27960C1-20 (i27960c1)
27960C2-25 (i27960c2)
27960C2-33 (i27960c2)
27960K1-16 (i27960k1)
27960K1-20 (i27960k1)
27960K2-20 (i27960k2)
27960K2-25 (i27960k2)
Library: /software/synopsys/1998.02/libraries/smartmodel
2159 models, 12415 timing-versions found
```

Figure 2.4 Library browser window

2. In the VHDL source program where smart model library components are to be used, we have to make the smart model library visible. For that, we add the following line at the start of the program:

```
Library SMARTMODEL;
Use SMARTMODEL.COMPONENTS.ALL;
```

3. After finding the required component in the smart model library, we then use it in the VHDL program. The smart model library contains a file "components.vhd" that contains the VHDL entity level description of all the available components. From the entity specification, we can obtain the input-output signals and generics associated with a component.
4. In our VHDL program, we define an entity which has the same number and type of I/O pins


String Search: *64

Vendors:
Intel
LMC
Lattice
LogicDev
Lucent
MIPS
MMI

Function/Subfunction:
MEMORIES: FIFO
MEMORIES: NVRAM
MEMORIES: ROM
MEMORIES: SRAM
PROCESSORS
PROCESSORS: DSP

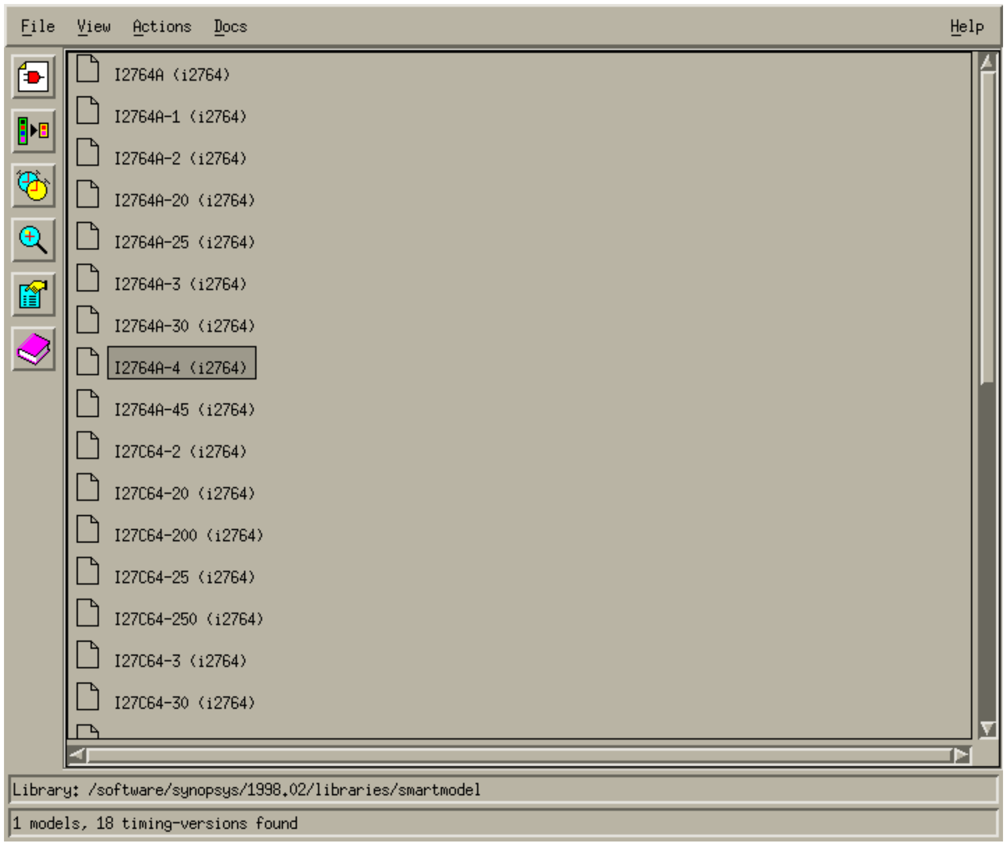
Licensed Packages:
dynamicty-all
faultsim-all
logicmodel-a1010_100
logicmodel-a1020_100
logicmodel-a1225_100
logicmodel-a1240_172
logicmodel-a1240_144

Summary of Filter Options:
(*64) AND (Intel) AND (MEMORIES: ROM)

Figure 2.5 Filter dialog box

as the model in the smart model library. We then instantiate this entity in the architecture of the main program. In the configuration section, we bind this instantiated component to the architecture of the smart model component.

In our project, since we are modeling the whole system in SGE, we first need to make a symbol for every smart model component. We then define the input-output pins whose number and type match those of the corresponding smart model component. When we extract the VHDL file from the symbol, it gives us the component entity with our supplied pin names and types, empty architecture body, and empty configuration. We then instantiate the component in the architecture. In the configuration, we bind this component with the architecture of the corresponding smart model component. We explain this process with an example:



```
I2764A (i2764)
I2764A-1 (i2764)
I2764A-2 (i2764)
I2764A-20 (i2764)
I2764A-25 (i2764)
I2764A-3 (i2764)
I2764A-30 (i2764)
I2764A-4 (i2764)
I2764A-45 (i2764)
I27C64-2 (i2764)
I27C64-20 (i2764)
I27C64-200 (i2764)
I27C64-25 (i2764)
I27C64-250 (i2764)
I27C64-3 (i2764)
I27C64-30 (i2764)
```

```
Library: /software/synopsys/1998.02/libraries/smartmodel
1 models, 18 timing-versions found
```

Figure 2.6 Results of the filtering process

Suppose we want to use the 74573 latch in our system. First we invoke the smart model library browser and in the string option of the filter dialog box, we enter 573. All the model names having the string 573 appear. Suppose we decide to select the model for SN74AS573.

We then go to the components.vhd file and find the entity description for SN74AS573. We get the following description:

```
component ttl573
               --Available timing versions:
               --54F573-FAI
               --74F573-FAI
               --74F573-SIG
               --74HC573-SIG
               --74HCT573-SIG
               --CD54AC573
               --CD54ACT573
               --CD54HC573
               --CD54HCT573
               --CD74AC573-EXT
               --CD74ACT573
               --CD74AC573
               --CD74ACT573-EXT
               --CD74HC573
               --CD74HCT573
               --IDT54AHCT573
               --IDT54FCT573
               --IDT54FCT573A
               --IDT74AHCT573
               --IDT74FCT573
               --IDT74FCT573A
               --SN54ALS573B
```

```
          --SN54AS573
          --SN54F573
          --SN54ABT573
          --SN74ALS573B
          --SN74AS573
          --SN74F573
          --SN74ABT573
     generic (
          TimingVersion : STRING := “SN74AS573”;
          DelayRange : STRING := “MAX”;
          ModelMapVersion : STRING := “01003”);
     port (
          D0 : in STD_LOGIC;
          D1 : in STD_LOGIC;
          D2 : in STD_LOGIC;
          D3 : in STD_LOGIC;
          D4 : in STD_LOGIC;
          D5 : in STD_LOGIC;
          D6 : in STD_LOGIC;
          D7 : in STD_LOGIC;
          LE : in STD_LOGIC;
          OE : in STD_LOGIC;
          Y0 : out STD_LOGIC;
          Y1 : out STD_LOGIC;
          Y2 : out STD_LOGIC;
          Y3 : out STD_LOGIC;
          Y4 : out STD_LOGIC;
          Y5 : out STD_LOGIC;
          Y6 : out STD_LOGIC;
          Y7 : out STD_LOGIC);
end component;
```

We see here that there are many components that have the same input-output pins and perform the same function, but can differ in timings and delays. The different versions differ from each other in timings, whereas for the same version, we can choose to simulate for maximum, normal or minimum pin to pin delay. We can put any of the available options in the generics. Here we are putting SN74AS573 for the timing version, MAX (maximum) for pin to pin delay and the given value for the model version. After setting the generics, we need to reanalyze the components.vhd file so that changes in the generics take effect. But the preferred approach is to leave the library file unchanged and change the generics during the component instantiation.

Next we make a symbol for 74573 with the same I/O pins as the smart model ttl573. The VHDL output of this symbol is as follows:

```
--VHDL Model Created from SGE Symbol hc573.sym – Mar 19 11:02:20
1998

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;

entity HC573 is
      Port (       D : In    std_logic_vector (7 downto 0);
                  LE : In    std_logic;
                  OE : In    std_logic;
                   Y : Out   std_logic_vector (7 downto 0) );
end HC573;

architecture STRUCTURAL of HC573 is

begin
```

```
end STRUCTURAL;

configuration CFG_HC573_STRUCTURAL of HC573 is
   for STRUCTURAL

   end for;
end CFG_HC573_STRUCTURAL;
```

We fill the empty architecture and configuration bodies and also make the smart model library visible. The complete VHDL file for the component thus is :

```
-- VHDL Model Created from SGE Symbol hc573.sym – Mar 19
11:02:20 1998

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;

library SMARTMODEL;
use SMARTMODEL.COMPONENTS.ALL;

entity HC573 is
      Port (       D : In    std_logic_vector (7 downto 0);
                  LE : In    std_logic;
                  OE : In    std_logic;
                   Y : Out   std_logic_vector (7 downto 0) );
end HC573;

architecture STRUCTURAL of HC573 is
```

```
begin
     U1: ttl573
     generic map(
          TimingVersion => “SN74AS573”,
          DelayRange => “MAX”,
          ModelMapVersion => “01003”)
     port map(
               D(0),D(1),D(2),D(3),D(4),D(5),D(6),D(7),
               LE,OE,
               Y(0),Y(1),Y(2),Y(3),Y(4),Y(5),Y(6),Y(7) );
end STRUCTURAL;

configuration CFG_HC573_STRUCTURAL of HC573 is
   for STRUCTURAL
      for U1: ttl573
        use entity SMARTMODEL.ttl573(SMARTMODEL);
      end for;
   end for;

end CFG_HC573_STRUCTURAL;
```

We apply the same procedure to all the components and then interconnect all the symbols in the SGE schematic editor to create the model for the whole system. The complete model of the system is shown in Figure 2.7

2.3.2 Additional VHDL models

For the purpose of simulation, and also for good observability of signals during simulation, the following additional VHDL models are included in the system:

The component *Control* is used for assigning permanent ‘1’ or ‘0’ to some pins of the microcontroller. For example, to set the mode as expanded mode, the pins MODA and MODB of the microcontroller must both be made ‘1’. Also, the interrupt pins XIRQ and IRQ should be made ‘1’ since they are not used here. The OE pin of the 74573 latch had to be made ‘0’.

The component *Connector* connects the 8-bit bus A(15:8) to 8 separate pins. By giving its input the name AD(15:8), we can then combine it with the multiplexed bus AD(7:0) so we can trace the complete address A(15:0) as a single signal on the waveform viewer of the VHDL simulator.

The reset of the 8255 is of opposite logic value to the reset of the microcontroller. So, a component *inverter* is used which is connected between the reset pin of the microcontroller and the 8255.

During simulation, we enter the analog signal value from the standard input i.e. keyboard. To take this input and give it to the ADC, we have made the model *Analog_Therm2.* This component is activated from the line 0 of PORTC. It operates in Text I/O mode of VHDL and after activation, takes the input real number and passes it to the ADC through its output port of type real. The VHDL source listing of the main program and all the components is given in Appendix C.

2.3.3 Memory image file (.MIF)

Memory components in the Smartmodel Library use the concept of a memory image file to load the memory content at the beginning of simulation. Each line of the .mif file contains the address and data for one memory location. Address and data must be separated by a slash; whereas at the end of the data, a semicolon is required. An example .mif for an 8 KB memory is given below:

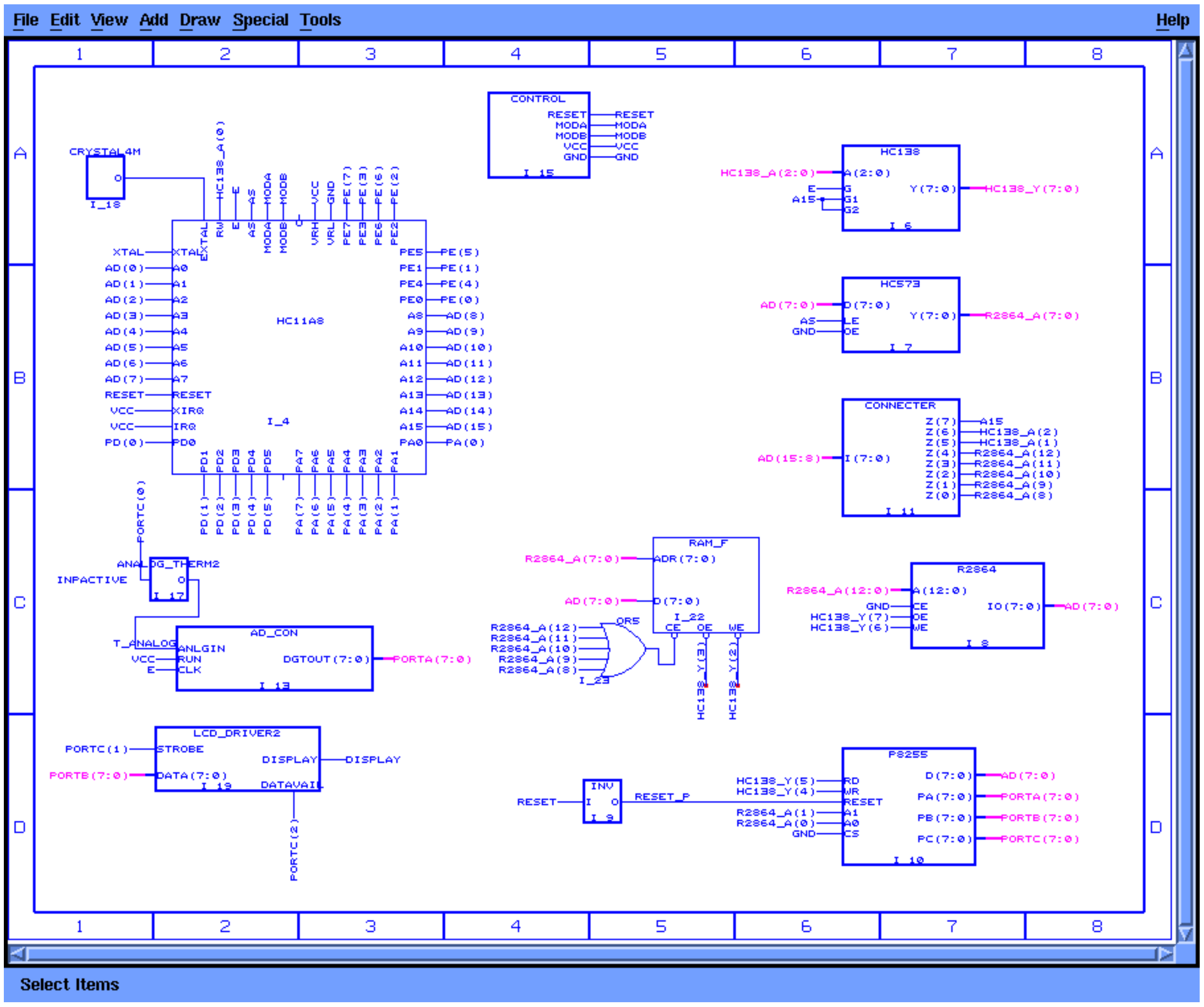


Figure 2.7 Complete model of temperature control system

```
# Memory image file for an 8 KB memory
0000/1A;
0001/2C;
0002/34;
0003/55;
.
.
1FFD/22;
1FFE/FB;
```

```
1FFF/38;
```

A # symbol at the start of a line denotes a comment. The data and address are assumed to be in hexadecimal form.

To convert an HC11 assembly program to Memory Image File format, two programs are needed, CASM11.EXE and IMAGE.EXE. CASM11 is a commercial program used for converting assembly to object code while IMAGE is written by Jason Liu in C to convert the object code to a .mif file. Both of these programs run on a PC. Therefore, the assembly program must be assembled and converted to .mif file on a PC.

In our temperature monitoring system model, the assembly program must be loaded into the external 2864 EEPROM and the interrupt vectors must be loaded into the internal ROM of the 68HC11. The translator program IMAGE translates the executable file (.s19) produced by the CASM11 assembler into memory image file format. Two files are produced upon execution of *image.exe*, memory.mif and rom.mif. Memory contents at address $6000 - $7FFF are stored in rom.mif; all others are stored in memory.mif. The interrupt vector addresses are located in the 8 KB ROM which is internal to the microcontroller. These addresses are loaded with interrupt vectors by the file "memory.mif". Also when the system powers up, it starts its operation from the location E000, which is the first address of the internal ROM. Since we are using the external EEPROM as the storage location for our system program, we write the opcode of the jump instruction at E000 which then jumps to the external EEPROM for executing program instructions. This opcode at E000 is also written in the "memory.mif" file. At the beginning of the simulation, EEPROM 2864 loads memory from rom.mif; and 68HC11A8 loads memory from memory.mif. The translator *image.exe* is specifically written for the temperature monitor model. If the external EEPROM is mapped into different addresses or there are more external ROM or RAM, the translator must be modified. The source file image.cpp is included in appendix B.

The syntax for running the assembler is:

```
casm11 <filename.ASM>   l  s
```

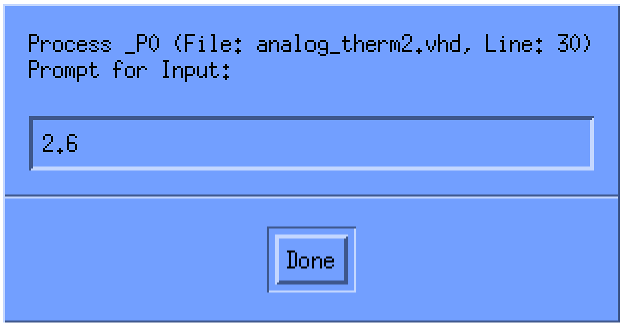
With the option ‘s’, we are telling the assembler to form the object code in Motorola hex format. The resulting file has the extension .s19. With the option ‘l’ (lower case L), we are telling the assembler to make a list file (.lst) also which contains the object code as well as source listing of the code.

The syntax for running the image program is

```
image <filename.s19>
```

This gives us the two files, rom.mif and memory.mif as described above. After analyzing the system model, we then have to simulate it using the VHDL simulator/debugger. We need to transfer the .mif files from the PC to the Sun Sparc platform before starting the simulation.

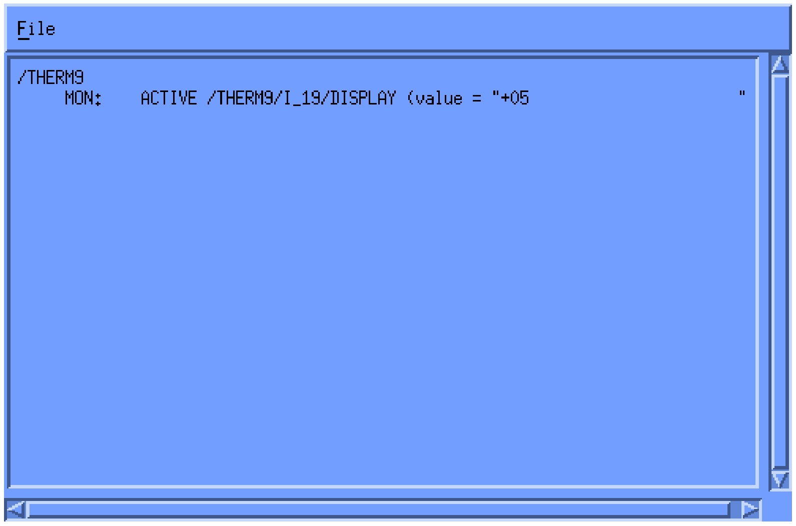
A control file was written in Synopsys Simulation Control Language (SCL) to aid the simulation process. The control file specified the signals to be traced and generated an output window to show the results. These input and output windows are shown in Figures 2.8 And 2.9 respectively. The control file is given in appendix C.17



Process _P0 (File: analog_therm2.vhd, Line: 30)
Prompt for Input:

2.6

Figure 2.8 Input window



```
/THERM9
     MON:    ACTIVE /THERM9/I_19/DISPLAY (value = "+05                         "
```

Figure 2.9 Output window

2.4 Hardware implementation of the system

In the second phase of the project, the system was actually implemented in hardware. This implementation was done on the 68HC11 EVBU trainer kit and prototyping strips. The system was connected to a host PC through a serial port. The host software PCBUG11 was used to control, monitor and debug the activities on the system. The following sequence of steps were taken :

(1) Build the system using 68HC11 EVBU, external chips for EEPROM, latch, decoder, PPI, display and control, and the prototyping wires

(2) Connect the system to the PC through the serial port

(3) Set the jumper settings on the system corresponding to the special bootstrap mode of the microcontroller. In the bootstrap mode, the system can read in the program code from the outside using the serial communications interface of the microcontroller [SpasovP96]. The PCBUG11 software communicates with the system in the bootstrap mode only.

(4) Turn on the system

(5) Run the PCBUG11 software from the host

(6) Whenever the 68HC11 powers up, it starts executing instructions from the internal location $B600. Since we are using the external EEPROM for program storage, we need a jump to the starting address of the external EEPROM from this location. To do this, we first need to define the internal EEPROM locations and then unprotect them i.e. make writing on them possible. We use the following commands from PCBUG11:

```
MS        $1035     $10
EEPROM    $B600     $B7FF
```

(7) Write the instruction for jump to external memory location at the location $B600.

```
ASM       $B600
JMP       $6000
```

(8) In the bootstrap mode, the microcontroller operates in a stand alone mode and does not recognize the external circuitry attached to it. To load and run the program on external EEPROM, we first need to define the external EEPROM locations and then change the operating mode of the microcontroller to expanded mode. Use the following commands:

```
  MS      $103C     $26
 EEPROM   $6000     $7FFF
```

(9) Load the object code file (.S19) of the system on the external EEPROM using the command:

```
LOADS     FILENAME.S19
```

(10) Run this program from the system using the command:

```
G         $6000
```

Steps (6) and (7) can also be done in another way. We can define a macro and put the PCBUG11 instructions in it and can then execute them all at once. The following macro was written for the sytem:

```
DEFM INIT
BEGIN
```

```
        MS $1035 $10
        EEPROM $B600 $B7FF
        MS $103C $26
        EEPROM $6000 $7FFF
    END
```

Run the macro using the following sequence of commands:

```
LOADM INIT.MCR
INIT
```

Once the macro is loaded, we can then load a program into memory and run it.

To run the program previously loaded into the external EEPROM, we just need to change the jumper settings on the EVBU corresponding to the expanded mode. Then we power on the board. The system will start from $B600 and then jump to the system program at external location $6000.