ENGI E1102 Departmental Project Report: Computer Science/Computer Engineering

Pranav Shrestha, Jeffrey Wan, Ravi Jalan

Fall, 2016

Abstract

Studying computer science, it is imperative to understand how our software is ultimately works at the lowest level - hardware. This hardware-software link has been a mystery to many software engineers who seldom have the chance to develop this intuition, simply because they have not been exposed to the interaction between electrical engineering and computer science. For non-coders, this project serves as a jumpstart into the complex world of computer science, by presenting a easy-to-approach, tangible interpretation of computer science.

This report outlines the various steps involved in writing calculator functionality for the HP20b in the language C. Our final product is capable of understanding Reverse Hungarian Notation, a simple syntax that is natural to program. This report looks into the challenges involved in each of these steps and how these issues were resolved. Finally, This report provides a background knowledge on the HP20b calculator in terms of the processor, LEDs, as well as keyboard.

1 Introduction

Despite the surge in computational power and ubiquity of processor chips over the years, the need for handheld calculators is still evident. As with any programmable device, handheld calculators are a perfect examples of real-world uses of embedded programming. Programming such a system involves addressing challenges from power consumption to efficient algorithmic design and mathematical notation.

The HP 20b business consultant is a programmable financial calculator produced by Hewlett-Packard (circa 2008). It has 2 lines of display. The first line had a 6x42 pixels and 11 indicator signs and the second line was used a 12+3 digit display. The calculator's processor's serial interface could be accessed from below the battery cover to allow updating the firmware through a special HP cable.

This project aims to program a factory-wiped version of this calculator into a functional Reverse Polish Notation (RPN) calculator. The use of RPN in HP 20b programming can be attributed to its postfix notation system, where every operator follows its operand thereby removing operational ambiguity and consequently, the need for parentheses.

2 User Guide

The calculator understands Reverse Polish Notation, a notation where all the operands precede the operators.

For example, in order to add 1 and 2, one would type 1 2 + instead of 1 + 2. To multiply 1 and 2, type 1 2 x.

More complex expressions equations can be understood using one simple rule: when an operator is read from the keyboard, the calculator evaluates the two operands that directly precede it.

For 3 4 5 x -, When the first operator x, is pressed three numbers have already been inputted, 3, 4, and 5. In Reverse Polish Notation, 4 and 5 are evaluated, leaving 4 x 5 = 20. This leaves 3 and 20 as the two numbers remaining. Our last operator, -, operates on the 3 and 20, returning 3 - 20 = -17.

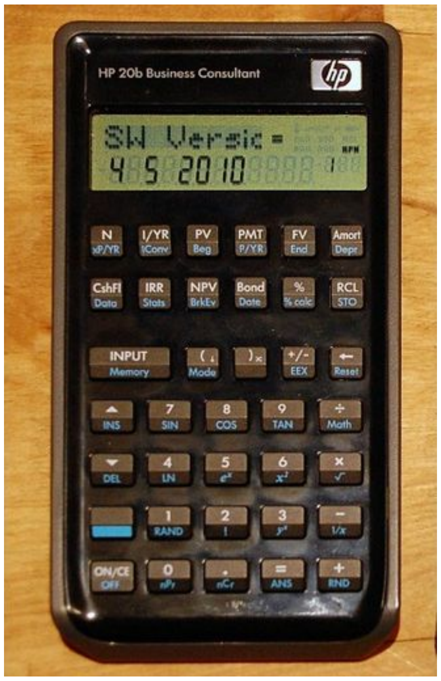

Figure 1: The HP 20b

3 The Platform

The HP 20b calculator used in the project was originally published by Hewlett-Packard as a Business Consultant. A detail analysis of the circuitry and the schematics of the calculator were presented in the first lab assignment. A more detailed diagram was obtained from the online manual. More information was obtained from the HP 20b repurposing project online resource [1].

3.1 The Processor

The hardware of the calculator may seem like its most important parts as we directly interact with it, however, what keeps the clock ticking and the calculator functioning is the processor.

The AT91SAM7L128 is a low power member of Atmelś Smart ARM microcontroller family based on the 32−bit ARM7TM RISC processor and high-speed Flash memory. AT91SAM7L128 features a 128 Kbyte high−speed flash memory and a total of 6 Kbytes SRAM. This is important as it emphasis on low power and power conservation as this unique design which contains the system controller. The system design uses a software to control the power output and the clock of each peripheral individually that gives it an extraordinary advantage of saving power and energy by shutting of the power supply of the peripherals that are not in use. This may however cause a problem incase the peripheral is being accessed as it may seem like the system is not functioning correctly.

The standard processor is surrounded by multiple peripherals that have circuitry running to and fro the processor (Figure 2), which also allows the processor to change the power supply and clock functionalities of the peripherals. The processor may look very intricate but it is in laymanâĂŹs term the central processing chip surrounded by the memory and the peripheral systems, many of which were not used during the lab.

3.2 The LCD Display

The LCD Display contains 2 lines of alphanumeric LCD display in which the first line contains eight characters of scrolling display with an additional eleven indicators. The second line has twelve major digit displays along with three additional digit displays. The contrast in the whole screen is adjustable. The LCD contains of multiple pixels that are controlled by individual bits. These bits can be manipulated to access different memory locations and toggle between different pixels. Each pixel is connected to different wires that are arranged in ascending rows and columns in the second line that includes the major display section. These can be tweaked easily to form different alphanumeric characters. However to the left of the first line is a grid of pixels arranged in a peculiar

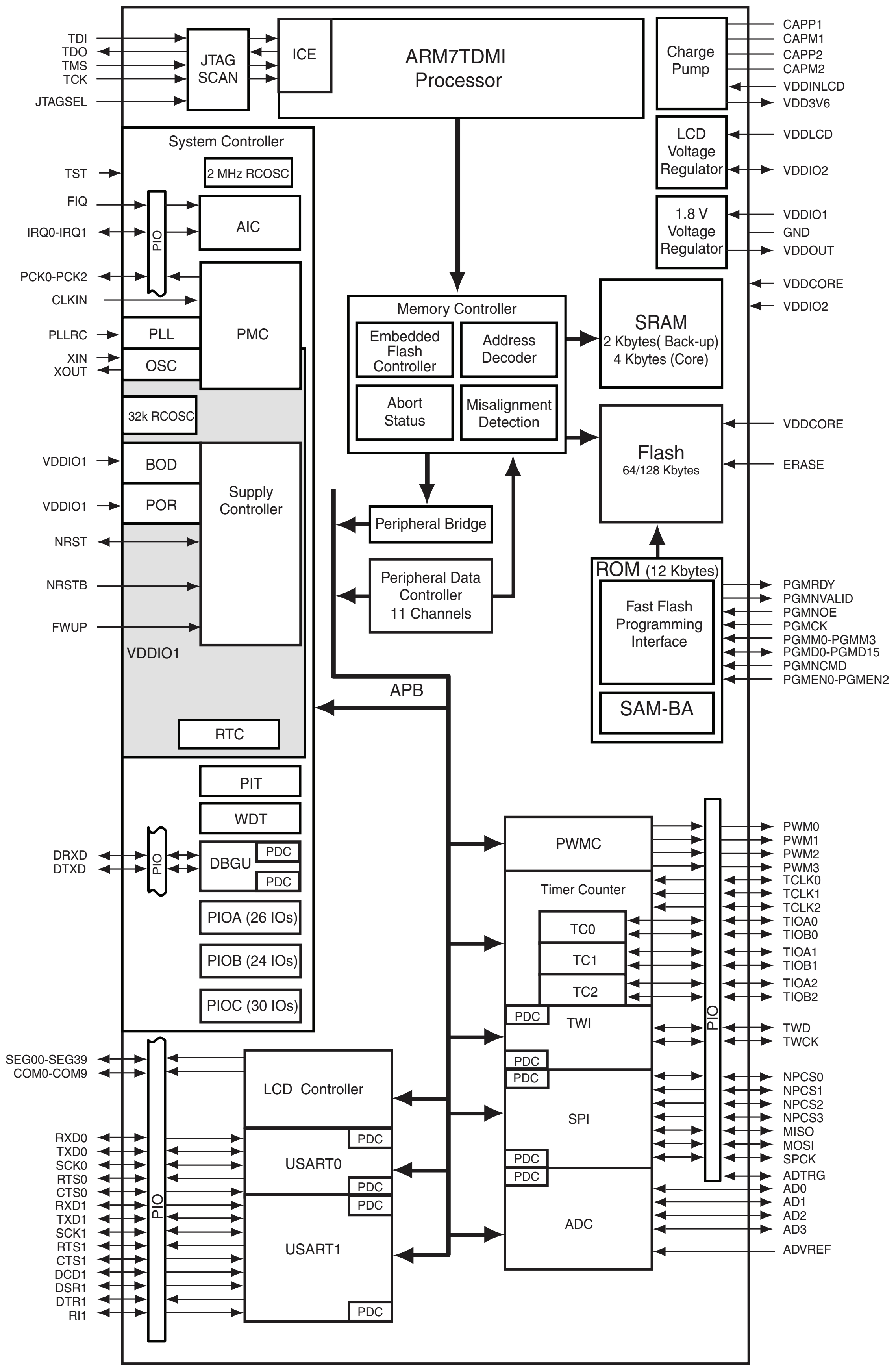

Figure 2: A block diagram of the AT91SAM7L microcontroller that is at the heart of the HP 20b

manner. It has a section that has regular rows and columns accompanied by a rotated section of the pixels that has a 90 degree counter clockwise rotation. Therefore to access the grid the circuitry should be studied in details.

Functions such as display `void lcd_put_char7 (char ch, int col)` display an ASCII character in the given column on the 7-segment display. The character ch and the column it needs to be printed in should be specified in order for the display to depict the integer. Other functions similar to this were used to display pixels in the top band of pixels.

3.3 The Keyboard

Although it may seem very utilitarian, the interior of the keyboard has an ingenious design. When we tore apart some old keyboards during class we discovered that they keyboard had three layers. Two silicon sheets that contained detailed circuitry that completed the loop and a this nonconductor sheet in the middle that had holes where the keyboard keys were being pressed. When the keys of the keyboard were pressed the silicon pads would be pressed through the holes in the nonconductor and would complete the circuit, informing the processor about the key being pressed and the signal would be completed. The HP 20bś keyboard worked in a similar manner.

4 Software Architecture

The program was primarily coded in C. Various definition files were made available, including files for compilation, memory definitions and flashing. These include:

Makefile	Compilation rules for the program, including source file paths to compile
flash.bat	Batch file code to compile the code and flash it into the HP 20b
flash-h.bat	Batch file variant for ARM-USB-TINY-H dongles
AT91SAM7L128.h	Addresses of every peripheral in the AT91SAM7L128 chip
crto.S	ARM assembly code that initializes the C runtime environment
AT91SAM7L128.lds	Linker script defining where and how to put the program in memory
flash.cfg	OpenOCD file: rules for writing to flash
hp-20b-calculator.cfg	Configures chipname to SAM7L chip for the OpenOCD
AT91SAM7L128.cfg	Configures information about the SAM7L chip for the OpenOCD

Table 1: System files for flashing embedded firmware into HP 20b calculator

The main processing for the implementation of the calculator itself rests in the following files:

keyboard.c	Initializes, handles and parses user input from the keyboard
main.c	Handles program flow, processing user input and displaying processed results
lcd.c	Initializes and handles LCD screed output including numbers, characters and pixels
keyboard.h	Externally visible interface to keyboard.c
lcd.h	Externally visible interface to lcd.c

Table 2: C files for implementation of the RPN calculator

5 Software Details

This section deals with the actual implementation of the code written. Each subsection defines an integral step toward implementing a fully functional RPN calculator.

5.1 Lab 1: Display a Number

The aim of this part of the project was to edit the pre-existing code to create a function that takes an integer argument and displays it in decimal on the calculator.

The LCD display of the calculator has 15 placeholders for digits (12 normal digits and 3 superscripted ones). The assignment was to display positive or negative integers using the 12 digit placeholders on the screen display.

Listing 1: main.c to display a number

```c
#include "AT91SAM7L128.h"
#include "lcd.h"

int main()
{
  lcd_init();
  display_a_number(2);
  return 0;
}

void display_a_number(int a){
  int column=11;
  if(a==0)
    lcd_put_char7('0',11);
  else{
    int isnegative = 0;
    if(a<0){
      isnegative = 1;
      a = -a;
    }
    while(a>0){
      lcd_put_char7(((a%10)+48), column--);
      a=a/10;
    }
    if(isnegative==1)
      lcd_put_char7('-',column);
  }
}
```

The main code displayed in Listing 1 initializes the LCD and calls the display function. The display function first checks for a zero, and if the number is 0, then it displays a zero and exits. Then it checks if the number is negative, and proceeds to display the absolute value of the number. It does this by printing the lowest digit of the number in the rightmost column, then every other digit in the suceeding columns until all digits have been displayed. Finally, it displays the '-' symbol if the number was negative.

5.2 Lab 2: Scanning the Keyboard

The assignment was to write software that reads the keyboard on the HP 20b and display the key being pressed by the user.

In the main code, after initialize the keyboard and lcds, we have a infinite loop that listens for what is being pressed by continually scanning the keyboard using the `keyboard_key` function in the keyboard code. If scanner detects that something is pressed, it would print out a number representing the key that was pressed.

To scan the keyboard, the `keyboard_key` function iterates through each column. For each column we first set the column to low and then iterate through all the rows. If the key that is pressed down is on that column and row, reading the row will return 0. If it found a key that is being pressed down, we reset the column to high and return the value of the 2-dimensional array at that row and column. (In the keyboard code, it has a 2-dimensional array mapping the rows and the columns to numbers.)

If none of the row readings returned 0, we reset this column to 0 and we proceed to the next column. If nothing was pressed after iterating through all the columns, the function returns -1.

Listing 2: main.c to scan the keyboard

```
#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"

int main()
{
 // Disable the watchdog timer
 *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;

 //initialize keyboard and lcd
 lcd_init();
 keyboard_init();

 //just keep listening for user input
  for(;;){
    //if something is pressed, print it
    if(keyboard_key()!=-1)
      lcd_print_int(keyboard_key());
    //print Not pressed!
    else
      lcd_print7("Not␣pressed!␣");
  }
}
```

Listing 3: keyboard.c to scan the keyboard

```
#include "AT91SAM7L128.h"

#define KEYBOARD_COLUMNS 0x7f
#define KEYBOARD_ROWS 0x400fc00

const unsigned char keyboard_row_index[] = {11,12,13,14,15,26};

void keyboard_init()
{
  // Initialize the keyboard: Columns are outputs, rows are inputs
  AT91C_BASE_PMC->PMC_PCER = (uint32) 1 << AT91C_ID_PIOC; // Turn on PIOC clock
  AT91C_BASE_PIOC->PIO_PER = KEYBOARD_ROWS | KEYBOARD_COLUMNS; // Enable control
  AT91C_BASE_PIOC->PIO_PPUDR = KEYBOARD_COLUMNS; // Disable pullups on columns
  AT91C_BASE_PIOC->PIO_OER = KEYBOARD_COLUMNS;   // Make columns outputs
  AT91C_BASE_PIOC->PIO_PPUER = KEYBOARD_ROWS;    // Enable pullups on rows
  AT91C_BASE_PIOC->PIO_ODR = KEYBOARD_ROWS;      // Make rows inputs

  AT91C_BASE_PIOC->PIO_SODR = KEYBOARD_COLUMNS;  // Drive all columns high
}

void keyboard_column_high(int column)
{
  AT91C_BASE_PIOC->PIO_SODR = 1 << column;
}
```

```
void keyboard_column_low(int column)
{
  AT91C_BASE_PIOC->PIO_CODR = 1 << column;
}

int keyboard_row_read(int row)
{
  return (AT91C_BASE_PIOC->PIO_PDSR) & (1 << keyboard_row_index[row]);
}

//2D Array of calc inputs
const int val[7][6] = {
  {10,11,12,13,14, 35},
  {15,16,17,18,19, 36},
  {20,21,22,23,24, -1},
  {25,7,8,9,26, -1},
  {27, 4,5,6,28, -1},
  {29, 1, 2, 3, 30, -1},
  {-1,0,31,32,33,34, -1}
 };

int keyboard_key()
{
  int col, row;
  //loop through each column
  for(col = 0; col<7; col++){
    //set this column to low
    keyboard_column_low(col);

    //check if any row is pressed in that column
    for(row = 0; row < 6; row++)
      if(!keyboard_row_read(row)){
        //reset column to high and return value of pressed key
        keyboard_column_high(col);
        return val[col][row];
      }

    //reset column to high
    keyboard_column_high(col);
  }

  //if nothing was pressed, return -1
  return -1;
}
```

5.3 Lab 3: Entering and Displaying Numbers

The assignment is to create a calculator that allows the user to display any positive or negative integer through the follow process: pressing a series of number keys, followed by an operator or input key. (The positive/negative key can be pressed at any time before the operator is pressed).

Our code utilizes a struct called entry, encapsulating the operation and number.

In the main code we declare our entry variable and then pass its address to the `keyboard_get_entry` function. The function sets the operator and number values to what the user has entered. Finally we print out the positive/negative number and the operator.

In the keyboard code, the `keyboard_get_entry` function. We have a loop that continually listens for keys, and only breaks when an operator is pressed.

As the series of number of keys is being entered, we can accumulate these individual digits into a number through the follow process: Initialize `myNum` to 0. Then, as numbers are entered, we add it to 10 multiplied by `myNum` and assign this to `myNum`.

If an operator is pressed we assign it to the operator of entry and break out of our loop.

If the positive/negative key was pressed, we flip the `isNegative` variable.

After breaking out of the loop, we can see whether our number is positive or negative based on whether the `isNegative` variable is 0 or 1. We can then assign the number of entry to be either the positive or negative `myNum`.

If no number is pressed before the operator is pressed the number our number as `INT_MAX` as specified by the use case. If someone presses the positive/negative key before entering any digits, we flip the `isNegative` variable and proceed normally.

Listing 4: main.c to display entered number

```c
#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"

int main()
{
  struct entry entry;
  // Disable the watchdog timer
  *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;

  lcd_init();
  keyboard_init();

  lcd_print7("PRESS");

  //Get entry
  keyboard_get_entry(&entry);

  //print negative or positive number
  if(entry.number<0)
    lcd_print_int_neg(1, -entry.number);
  else
    lcd_print_int(entry.number);

  //print operator
  lcd_put_char7(entry.operation,0);

  return 0;

}
```

Listing 5: keyboard.c to display entered number

```
#include "AT91SAM7L128.h"
#include "keyboard.h"

#define NUM_COLUMNS 7
#define NUM_ROWS 6
#define KEYBOARD_COLUMNS 0x7f
#define KEYBOARD_ROWS 0x400fc00

const unsigned char keyboard_row_index[] = {11,12,13,14,15,26};

/* Character codes returned by keyboard_key */

const char keyboard_keys[NUM_COLUMNS][NUM_ROWS] = {
  {'N',  'I', 'P', 'M', 'F', 'A'},
  {'C',  'R', 'V', 'B', '%', 'L'},
  {'\r', '(', ')', '~', '\b', 0},
  {'\v', '7', '8', '9', '/',  0},
  {'\n', '4', '5', '6', '*',  0},
  {'S',  '1', '2', '3', '-',  0},
  { 0,   '0', '.', '=', '+',  0}};

void keyboard_init()
{
  // Initialize the keyboard: Columns are outputs, rows are inputs
  AT91C_BASE_PMC->PMC_PCER = (uint32) 1 << AT91C_ID_PIOC; // Turn on PIOC clock
  AT91C_BASE_PIOC->PIO_PER = KEYBOARD_ROWS | KEYBOARD_COLUMNS; // Enable control
  AT91C_BASE_PIOC->PIO_PPUDR = KEYBOARD_COLUMNS; // Disable pullups on columns
  AT91C_BASE_PIOC->PIO_OER = KEYBOARD_COLUMNS;   // Make columns outputs
  AT91C_BASE_PIOC->PIO_PPUER = KEYBOARD_ROWS;    // Enable pullups on rows
  AT91C_BASE_PIOC->PIO_ODR = KEYBOARD_ROWS;      // Make rows inputs

  AT91C_BASE_PIOC->PIO_SODR = KEYBOARD_COLUMNS;  // Drive all columns high

}

void keyboard_column_high(int column)
{
  AT91C_BASE_PIOC->PIO_SODR = 1 << column;
}

void keyboard_column_low(int column)
{
  AT91C_BASE_PIOC->PIO_CODR = 1 << column;
}

int keyboard_row_read(int row)
{
  return (AT91C_BASE_PIOC->PIO_PDSR) & (1 << keyboard_row_index[row]);
```

```
}

int keyboard_key()
{
  int row, col;
  for (col = 0 ; col < NUM_COLUMNS ; col++) {
    keyboard_column_low(col);
    for (row = 0 ; row < NUM_ROWS ; row++)
      if (!keyboard_row_read(row)) {
    keyboard_column_high(col);
    return keyboard_keys[col][row];
      }
    keyboard_column_high(col);
  }
  return -1;
}

void keyboard_get_entry(struct entry *result)
{
  //initialize values for program to run
  char myOperator;
  result->number = INT_MAX; result->operation = 0;
  int myNum = 0; int isNegative = 0;

  for(;;){
    //set the input key as -1
    int myKey = -1;
    //print out the number as of now
    lcd_print_int_neg(isNegative, myNum);

    //get a new user input and store it in myKey
    while(myKey == -1) myKey = keyboard_key();

    //if myKey is a digit, add it to current number
    if(myKey>='0' && myKey <='9')
      myNum = myNum * 10 + (myKey - '0');
    //if myKey is +-, change sign of number
    else if(myKey == '~')
      isNegative = 1-isNegative;
    //break if myKey is anything else
    else{
      myOperator = myKey;
      break;
    }

    while(myKey != -1) myKey = keyboard_key();
  }

  //return value with sign
  if(isNegative == 1)
```

```
    result->number = -myNum;
  else
    result->number = myNum;
  //return operator
  result->operation = myOperator;
}
```

5.4 Lab 4: An RPN Calculator

We begin by initializing the lcd and keyboard.

For our stack we initialize an integer array of size 16. The size of the stack needs to stay below this maximum size. Numbers in the array that are not occupied by actual numbers are held by a placeholder: `INT_MAX`. The actual size of our stack occupied by numbers is tracked by the `stacklength` variable. As we remove items from the stack, we decrement the `stacklength` variable. As we insert items into the stack, we increment the `stacklength` variable.

We create an infinite loop, that allows the calculator to perform multiple calculations. We obtain an entry using `keyboard_get_entry` and we add this entry to our stack of numbers. If an operator is entered, we pop off the last entry in our stack, and replace the 2nd to last entry with the result of the operation; we use a switch statement to perform different operations depending on the operator input by the user.

If there are more operations than numbers, we output a syntax error. If we divide by zero, we output an division by zero error. Finally, if there is no error and the operation syntax was valid, then we print the number.

Listing 6: main.c for a functional RPN calculator

```
#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"

void print(int x){
  if(x < 0)
    lcd_print_int_neg(1, -x);
  else
    lcd_print_int(x);
}

int main()
{
  int i;
  struct entry entry;
  // Disable the watchdog timer
  *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;

  //Initialize lcd and keyboard
  lcd_init();
  keyboard_init();

  //Initialize stack size as 16 arbitrarily
  int STACK_SIZE = 16;

  //Initialize the stack array, and stack initial length;
  int stack[STACK_SIZE];
  int error;
  int stacklength;

  for(;;){
    //set up for an operation
    error = 0;
    stacklength = 0;

    //clear the stack
    for(i = 0; i < STACK_SIZE; i++) stack[i] = INT_MAX;

    //get first entry
    keyboard_get_entry(&entry);

    //exit as soon as operation is =
    while(entry.operation != '='){

      //if entry.number is not INT_MAX, add it to stack
      if(entry.number != INT_MAX)
        stack[stacklength++] = entry.number;

      //If operation is not INPUT
      if(entry.operation!='\r'){

        //If stack operation is valid
        if(stacklength>1){

          //pop off stack
          int a = stack[--stacklength];

          //reset stack value of popped element
          stack[stacklength] = INT_MAX;

          //get last value in stack
          int b = stack[stacklength-1];

          //do the operation and update stack value
          switch (entry.operation){
            case '+':
              stack[stacklength-1] = a+b; break;
            case '-':
              stack[stacklength-1] = b - a; break;
            case '*':
              stack[stacklength-1] = a * b; break;
            case '/':
              //DIV by 0 error
              if(a == 0){
                error = 2;
                break;
              }
              stack[stacklength-1] = b / a; break;
            default:
              error = 1;
          }
        }
        else{
          //set error since there is more operations than numbers
          error = 1;
        }
      }

      //if there is error, break
      if(error>0) break;

      //print if there is number to print
      if(stacklength>0) print(stack[stacklength-1]);

      //get new entry
      keyboard_get_entry(&entry);
    }

    //if there is no error and operation syntax was valid, print the number
    if(error == 0 && stacklength == 1){
      if(stack[0] <0)
        lcd_print_int_neg(1, -stack[0]);
      else
        lcd_print_int(stack[0]);
    }

    //else print the error
    else if(error == 1)
      lcd_print7("SYNTAX_ERROR");
    else
      lcd_print7("DIV_BY_ZERO!");
  }
  //end the program: Never called.
  return 0;
}
```

5.5 *Bonus: 2D PingPong*

We modified the lcd functions to print pixels into the dot-matrix portion of the LCD display. This was done by modifying inbuilt functions based on the schematic diagrams of the SEGMENT layer (Figure 4) and the COMMON layer (Figure 5). The code used to achieve this (implemented in the modified lcd.c) is given in 7.

Listing 7: lcd.c to print pixel in the dot-matrix display

```
//We modified the lcd_put_char7 function so it would print arbitrary pixels
void pprint(uint8 com, uint8 seg, uint8 mode){

    //gets memory location of pixels and "mask"
    //gets the segement pixels memory
    AT91_REG *common_p = AT91C_SLCDC_MEM + (seg * 2);
```

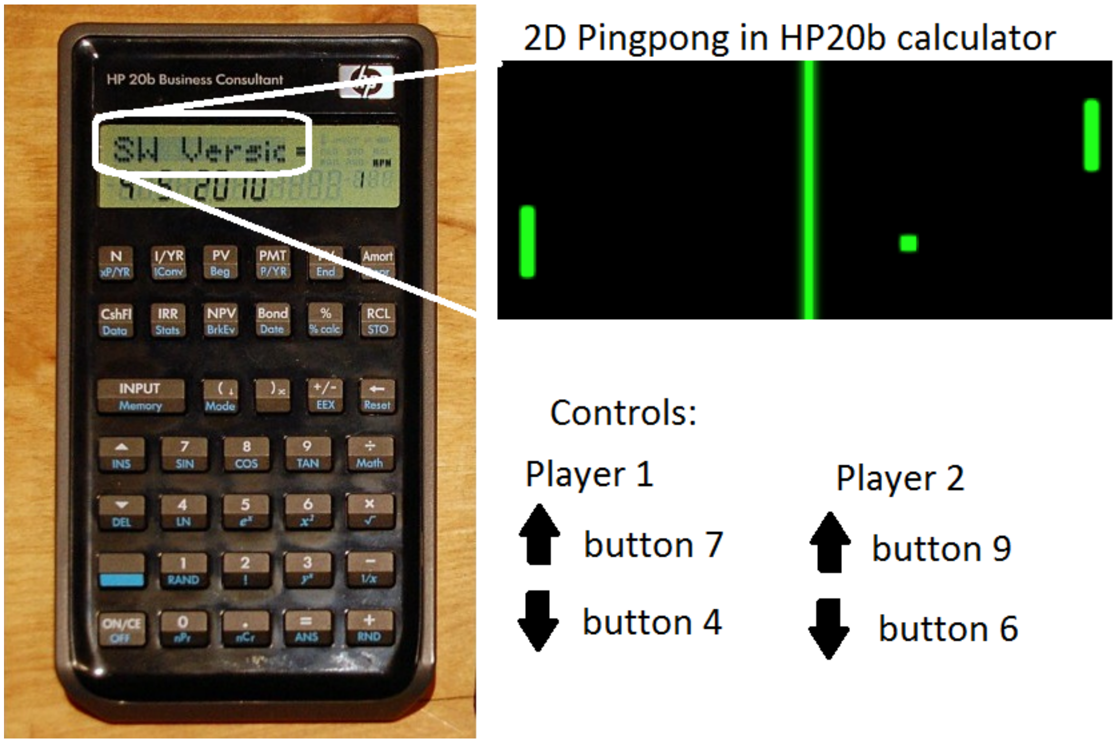

Figure 3: Simple PingPong game implemented in the HP 20b

```
    if (com > 31) common_p++;

    uint32 mask = ((uint32) 1) << (com & 0x1f);

    if(mode)
        //print out the pixel
        *common_p = *common_p | mask;
    else
        //removes the pixel. no idea why. because XOR = ~OR
        *common_p = *common_p ^ mask;
}

//Prints pixels based on a normal coordinate system.
//Since the last section of pixels, is rotated, we need this
//function to help translate from normal coodinates to how rows
//and columns are described in the lcd of the calculator
//mode defines whether to print or remove the pixel
void lcd_print(uint8 column, uint8 row, uint8 mode){
    //error checking
    if(column<0 || row < 0 || column > 42 || row > 5) return;

    if(column <33){
        if(row < 4) pprint(column + 6, row + 6, mode);
        else pprint(column + 6, row - 4, mode);
    }
    else if (column < 41){
        pprint(5 - row, column - 31, mode);
    }
    else{
        pprint(5 - row, column - 41, mode);
    }
}
```

After figuring out and implementing an efficient function to print out pixels at arbitrary (x,y) co-ordinates, implementing the pingpong game was a simple matter. It was implemented as shown in 8.

Listing 8: main.c to implement the PingPong game

```
#include "AT91SAM7L128.h"
#include "lcd.h"
#include "keyboard.h"

int main()
{
    // Disable the watchdog timer
    *AT91C_WDTC_WDMR = AT91C_WDTC_WDDIS;

    //Initialize lcd and keyboard
    lcd_init();
    keyboard_init();

    //Initialize delay
    const int delay = 100;
    int key;

    //Initialize scores
    int p1_score = 0;
    int p2_score = 0;

    //Initialize ball (x,y) speed
    int db_x = -1;
    int db_y = -1;

    //Print scoreboard
    lcd_put_char7('P', 0);
    lcd_put_char7('1', 1);
    lcd_put_char7('P', 7);
    lcd_put_char7('2', 8);

    //Play a game until score reaches five
    while(p1_score < 5 && p2_score <5){

        //Initialize paddle positions
        int p1_x = 1;
        int p1_y = 2;
        int p2_x = 41;
        int p2_y = 2;

        //Initialize ball position
        int b_x = 21;
        int b_y = 3;

        int scored = 0;

        //print score
        lcd_put_char7(p1_score+'0', 3);
        lcd_put_char7('-',4);
        lcd_put_char7(p2_score+'0', 5);

        //This is the physics loop, it handles all game logic
        while(scored == 0){
            //print paddles and ball
            int i;
            for(i = 0; i < 2; i++){
                lcd_print( p1_x, p1_y+i, 1);
                lcd_print( p2_x, p2_y+i, 1);
            }
            lcd_print( b_x, b_y, 1);

            //delay animation and get user input
            key = -1;
            for(i = 0; i < delay; i++){
                int tmpkey = keyboard_key();
                if(key == -1) key = tmpkey;
            }

            //Check collision with paddles
            //If ball hits left paddle
            if(b_x == 2 && (p1_y == b_y || p1_y + 1 == b_y) ) {
                //db_y stays the same
                db_x = -db_x;
            }
            //If ball hits right paddle
            else if(b_x == 40 && (p2_y == b_y || p2_y +1 == b_y) ) {
                //db_y stays the same
                db_x = -db_x;
            }

            //Check collision with side walls
            //If so, reset game and increase scores
            if(b_x == 0 || b_x == 42) {
                if (b_x == 0){
                    p2_score++;
                    scored = 1;
                }
                if (b_x == 42) {
                    p1_score++;
                    scored = 2;
                }
            }

            //Check collision with top wall or bottom wall
            if(b_y == 0 || b_y == 5) {
                db_y = -db_y;
            }

            //Else db_x, db_y stays the same

            //Clear the pixels of the paddles and balls.
            //Clear paddle
            for(i = 0; i < 2; i++){
                lcd_print(p1_x, p1_y+i, 0);
                lcd_print( p2_x,p2_y+i, 0);
            }
            //Clear ball
            lcd_print(b_x, b_y, 0);

            //Update paddle location using user input
            if(key == '9' && p2_y > 0) p2_y--;
            else if(key == '6' && p2_y <4) p2_y++;
            else if(key == '7' && p1_y >0) p1_y--;
            else if(key == '4' && p1_y <4) p1_y++;
            //Update the position of the ball based on its current velocity
            b_x += db_x;
            b_y += db_y;
        }

    }
    //When the score reaches 5, print winner message
    if(p1_score ==5 ) lcd_print7("CONGRATS␣P1!");
    else lcd_print7("CONGRATS␣P2!");
}
```

6 Lessons Learned

Computer science is a life skill required in this evolving technology based era. This lab provided us with a keen insight into the functioning of a basic calculator, a device every human uses unknowingly but doesnâĂŹt really have time to stop and thinking about the functionality of the calculator. The HP 20b was handed to us wiped cleaned and we were told to program a shell of hardware to a functioning calculator. What first seemed like an impossible task transformed to an engaging activity that revealed a lot about the functionality of the calculator.

It may seem like prior knowledge of programming would be sufficient to complete the task at hand, however this was not the case, an knowledge of hardware-software interface was also required. It may seem very trivial however there is a lot of planning and thinking that goes into programming a computer. A complete understanding of a wide array (pun intended), of data structures, using different functions, using controls intricately and manipulating the screen of the calculator.

The best advice to the future students would be to keep their code as clean as possible because it makes it understandable and makes it much easier to debug. Spending two hours debugging code only to realize that a statement is not in the correct loop taught us this lesson – the hard way.

We imagined successfully coding all the functions of the calculator by the end of the course, however we did not end up discovering and decoding the functionalities of all the keys. Although we wrote a program that took the users input, used all the operations and evaluated the result there is a lot more to learn. The amount of hard work we put in to program a mere calculator compels us to marvel at the intricacies of other technological milestones.

References

[1] Hp-20b repurposing project. Online `http://www.wiki4hp.com/doku.php?id=20b:repurposing_project`.

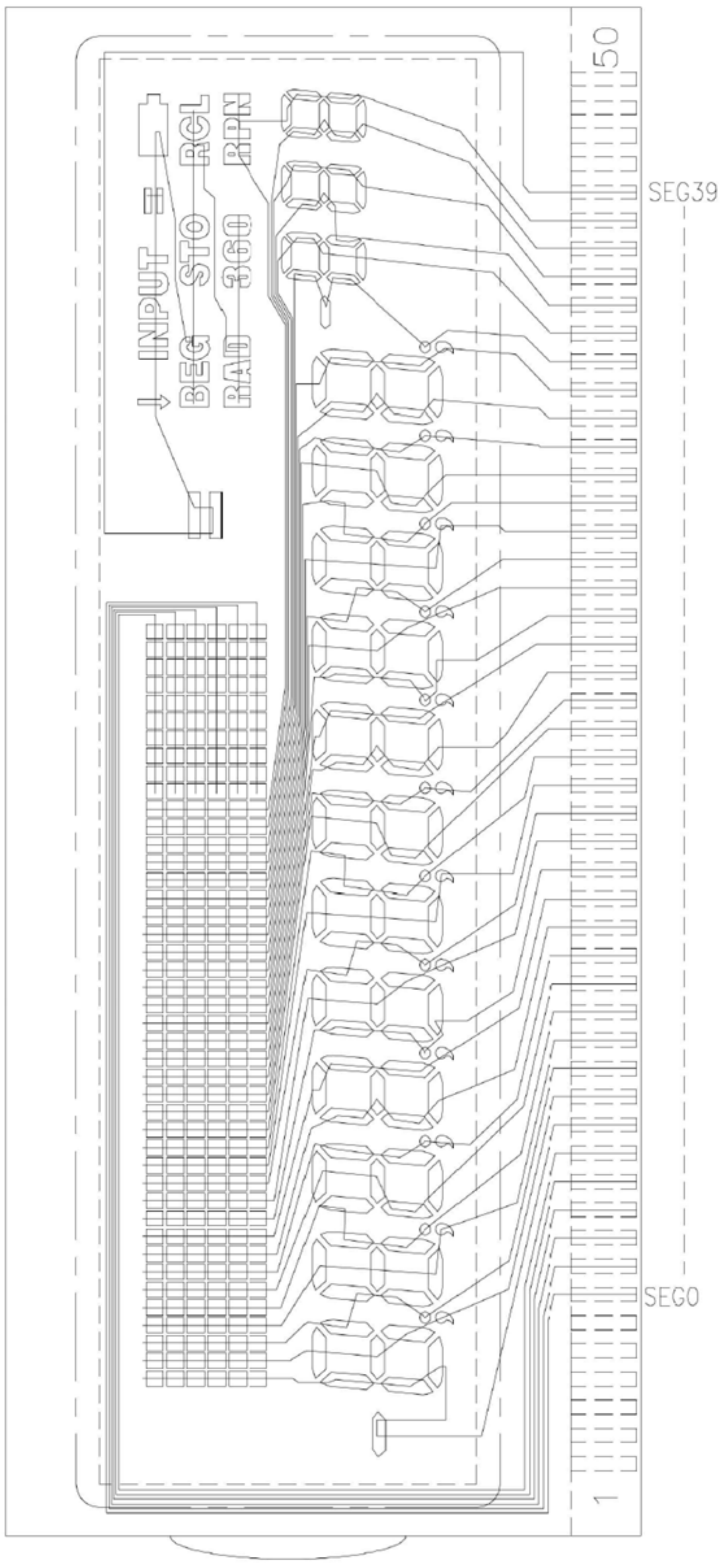

Figure 4: A schematic showing the top (SEGMENT) layer of the printed circuit board for the dot-matrix and segment display

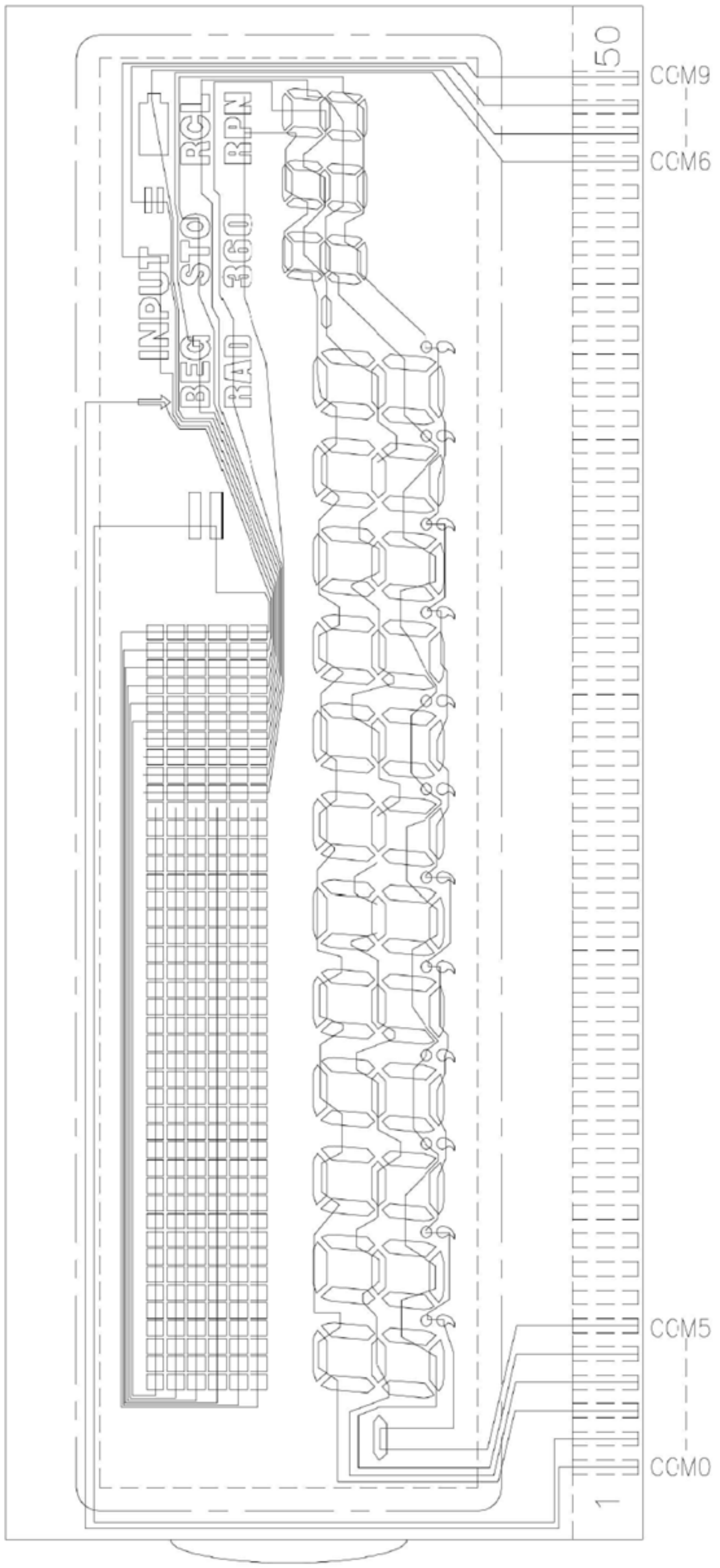

Figure 5: A schematic showing the bottom (COMMON) layer of the printed circuit board for the dot-matrix and segment display