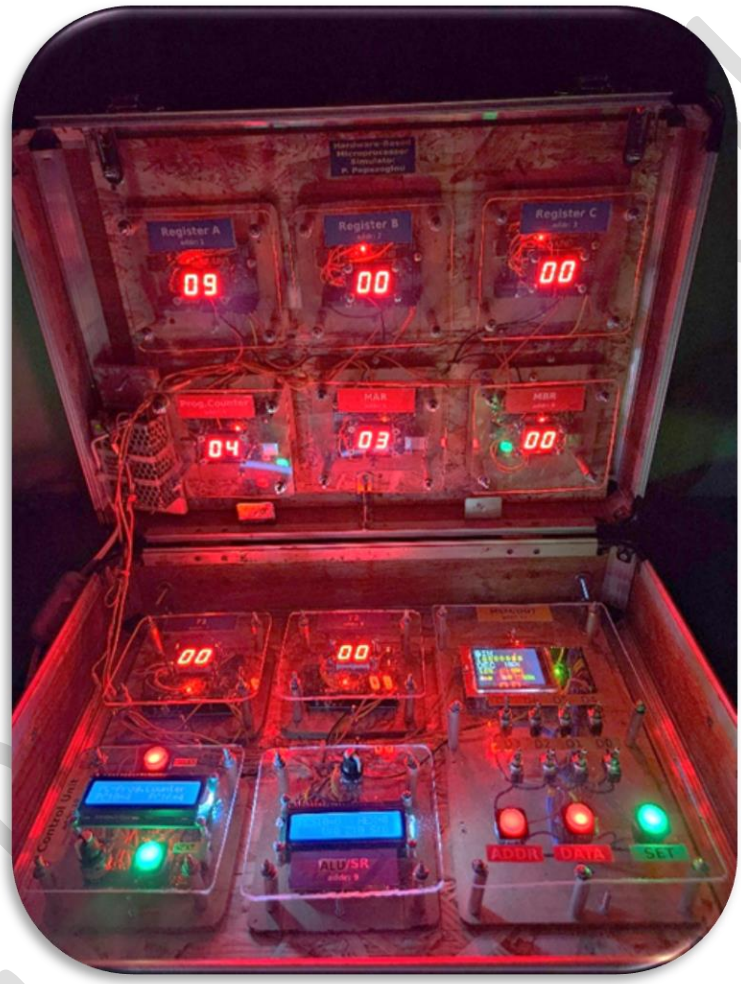
HARDWARE-ORIENTED MICROPROCESSOR SIMULATOR (HOMS)

OPEN-SOURCE PROJECT



Dr. Panayotis (Panos) Papazoglou

23 Feb – 2024 V1.0 (2023-2024)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material
The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for commercial purposes.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Contents

HARDWARE-ORIENTED MICROPROCESSOR SIMULATOR (HOMS)

Main features:

- full-working hardware-oriented simulator
- microprocessor/microcomputer simulator
- educational tool
- based on Arduino platform
- easy reproduction
- for academic teachers and researchers in the field of engineering education
- open-source

Keywords

Microprocessor simulator, Arduino-based educational tool, Hardware-oriented simulator, Open-source simulator

CHAPTER 1
General information

1.1 Introduction

The proposed HOMS system, constitutes a very different approach regarding the microprocessor simulation and faces effectively the corresponding limitations of the other relevant tools. Table 1.1 shows some differences between existing tools and the HOMS system.

Table 1.1

Feature	Software simulation	FPGA technology	HOMS system
Hardware point of view	NO	YES	YES
Hands on	NO	YES	YES
Complexity	LOW	HIGH	LOW
Architectural point of view (component level)	YES	NO	YES
Plug and Play	NO	NO	YES
Ease of use	YES	NO	YES
Custom architecture	NO	YES	YES
Custom assembly instructions	NO	YES	YES
Platform type	PC	PC and board	Board/Autonomous

Prior to HOMS system, a novel hybrid simulation platform has been proposed in the literature from the same author (Papazoglou, P., 2018). This platform is based on original designed PCBs with SMD technology. On the other hand, the above proposed implementation (Papazoglou, P., 2018) has limitations such as board assembly complexity, high cost, different board technologies and complex operation software. While the question for replacing microprocessor software simulators with hybrid approaches remains, a new educational tool for studying microprocessor architecture and programming has to be proposed for facing effectively all the previous tools limitations regarding construction, programming and operation complexity.

In this project, a fully working and mature educational tool for learning microprocessors is proposed for the very first time in the literature for higher education in the field of computer science and computer engineering. The proposed educational tool faces effectively every limitation of the previous versions, is based on open-source hardware and can be reproduced by everyone. Figure 1.1 shows the implementation of the proposed HOMS tool which is an 8bit microprocessor/system model. This model consists

of similar blocks that represent microprocessor internal components. The HOMS tool has also a memory/output unit for supporting memory data entry and data output. Is a full working custom system, where the corresponding developer can build its own assembly language and choose their desired microprocessor components.

Fig. 1.1 The proposed HOMS tool

CHAPTER 2
System description

2.1 Introduction

A microprocessor consists of various internal units for performing instruction execution. The internal units interact with each other by exchanging data. On the other hand, microprocessor fetches instructions from external memory and if needed, the corresponding results are returned to memory again. For building an operational microsystem (microcomputer), a microprocessor and a memory unit are needed as well as an input and an output unit. Figure 2.1 shows a typical model that represents a simple microprocessor.
Based on this model, the proposed hardware-oriented educational tool (HOMS) consists of the necessary units that form a simple microprocessor, a memory and supports data entry in memory as well as data output.

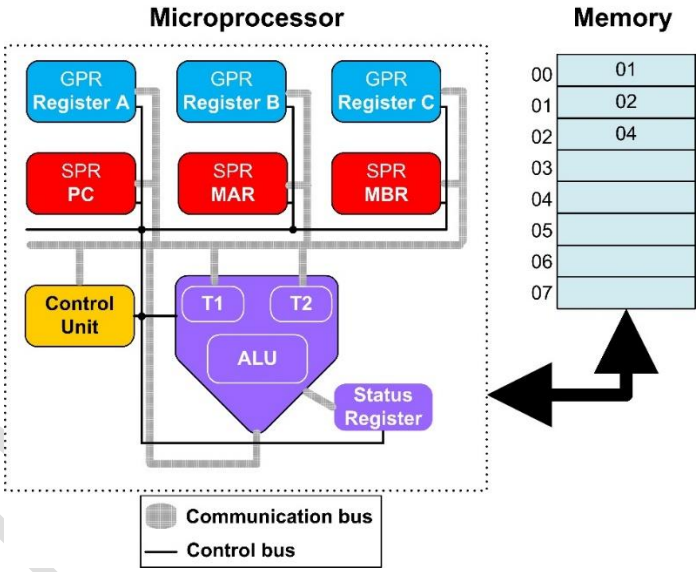


Fig. 2.1 Typical components of a single microprocessor

The main goal of a microprocessor is the instruction execution. Instructions are part of the program which is stored in main memory. Instructions have to be transferred from memory to microprocessor (fetching) via data bus. The instruction execution procedure is simplified in steps, as follows (assume that the addition A+B will be performed):

Step 1: Read PC (Program counter) for finding where the next instruction address is

Step 2: Store this address to MAR (Memory Access Register) in order to place the desired memory address on address bus

Step 3: Fetch instruction code via data bus and store this code in MBR (Memory Buffer register)

Step 4: Fetch parameter code via data bus and store this code in MBR (Memory Buffer register)

Step 5: Decode instruction data and execute, a) copy the contents of registers A and B to ALU registers T1 and T2 respectively, b) perform the addition T1+T2 and update SR (Status Register), c) store result to register A

Step 6: Update PC (Program Counter) for fetching next instruction code from memory

There is no hardware tool/kit in the market that shows the above procedure step by step. The proposed hardware-oriented tool (HOMS) consists of real hardware components that represent the microprocessor internal units, as shown in fig. 2.1.

2.2 HOMS tool Overview

The HOMS is a fully working prototype that offers unique features as compared to similar tools. Table 2.1 summarizes the supported features.

Table 2.1
HOMS tool unique features

Customizable architecture	The teacher or student can use any number or type of blocks for building the preferred microprocessor architecture.
Block reusability	The microprocessor units are based on the same board (e.g. Arduino UNO). For example, the proposed HOMS tool consists of eight identical blocks. Thus, the embedded software determines the block functionality (same block, different functionality based on software).
Programmable functionality	Based on the embedded software, a block operates like a register or control unit or ALU or special register, etc.
Experimental architecture	Based on the number and type of blocks, a teacher or student can test a prototype architecture or to expand an operation to smaller steps by using more blocks.
Assembly instructions development	The existing blocks support functionality that is controlled by the control unit. The HOMS user is free to build any assembly instruction which is supported by the software inside control unit.
Student skills emerge	Block building and assembly instructions can be developed by students. Students

	use their mind and hands and learn to perform accurate manipulations and movements to bring the hardware to life.
Educational scenarios	Using the default HOMS tool architecture, teachers can develop the desired assembly instructions for building different educational scenarios. Based on the embedded software (inside blocks), the behavior of the blocks can be adapted to the desired educational scenarios.
Complete approach	Using the proposed HOMS tool, the microprocessor can be approached from many different points of view/operation: a) a programmer/user develops and tests assembly code using exclusively the available system instructions b) a software/hardware developer adapts the embedded software for supporting the desired microprocessor functionality/operation c) a teacher or student selects the desired blocks and builds an entirely new architecture d) a student is simply watching/studying the instruction execution procedure/sequence
Multiple points of view	Based on the different points of view as mentioned previously, every student can extract information from the desired microprocessor feature.
Hardware point of view	The HOMS tool, emphasizes the hardware layer which is hidden in the existing simulation tools. Thus the “connection” of instruction, operation and hardware implementation is more clear in the student’s minds.
Standalone tool	The proposed HOMS tool does not need a PC and can be operated autonomously. Thus, constitutes a mobile laboratory system unit.
Easy reproduction	The hardware components of the HOMS tool can be found easily in any market. On the other hand, the multiple identical

	blocks support easily the reproduction procedure.
Open features	The main advantage of the implemented HOMS tool is the object-oriented approach and the open-source hardware which gives the freedom to any developer not only to reproduce the same tool but also to implement the whole simulator using different blocks (with or without an LCD, buttons, etc). Note that the embedded software makes the difference.

Based on the mentioned functionality and features, it is obvious that the proposed HOMS tool is more suitable for laboratory exercises in higher education in the field of computer science and engineering. ***Laboratory academic stuff may use the proposed prototype HOMS tool for building multiple boards and fully support a semester course. Moreover, the existing HOMS tool can be extended under the development of a thesis or a lab assignment or even under a research program for exploring new methods and tools in engineering education.***

2.3 Hardware components

Figure 2.2 shows the hardware block types that are used as internal microprocessor components and external memory. There are four types of blocks. Block-A is a general-purpose block and can be used for implementing registers and other microprocessor components. Blocks B and C just support different display capabilities but are also based on an Arduino UNO. Blocks A to C can be implemented as identical blocks using the same display module. The only different block is the block-D where an Arduino MEGA 2560 is used for supporting the LCD/TFT screen as well as switches and buttons.

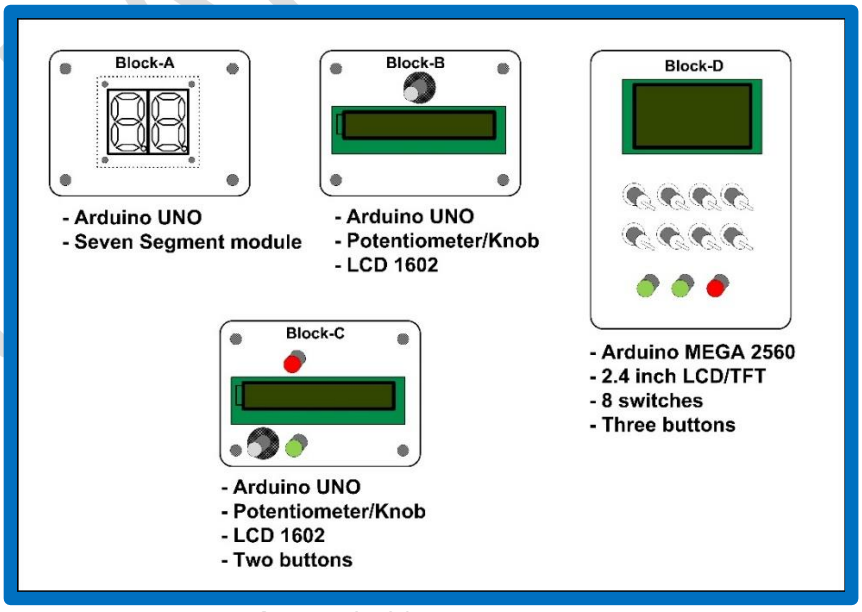


Fig.2.2 Block/Component types

Figure 2.3 shows the circuit connections between Arduino UNO and the seven-segment module as well as the physical implementation. The seven-segment module is based on two 74HC595 ICs. These ICs are shift registers with eight outputs each. For every digit to be displayed, 8bits are transferred serially and thus, few cables are used. When the 74HC595 buffer is full, the corresponding segments are updated. Due to 74HC595 ICs, no refresh is needed if the digits are not changed.

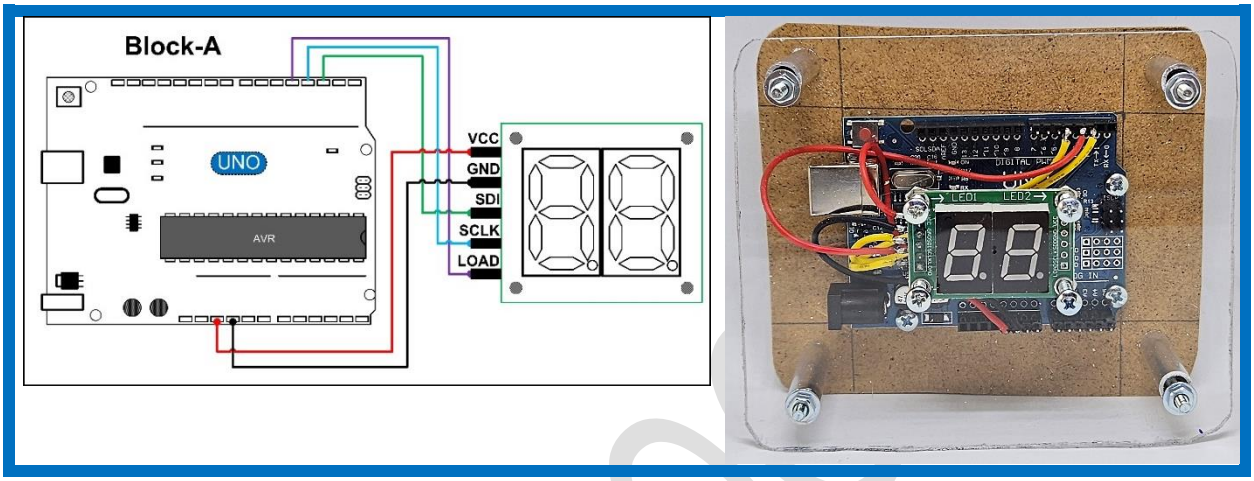


Fig. 2.3 (a) Block-A circuit and (b) physical implementation, code file: HOMS-REG.ino

Figures 2.4 to 2.6 show the circuits of block types B, C and D as well as the corresponding physical implementations.
All the blocks are implemented with very simple circuits and no special experience is needed. As shown in fig. 2.4, the block-B supports only an LCD 16x02 and a potentiometer which is used for controlling the LCD contrast.

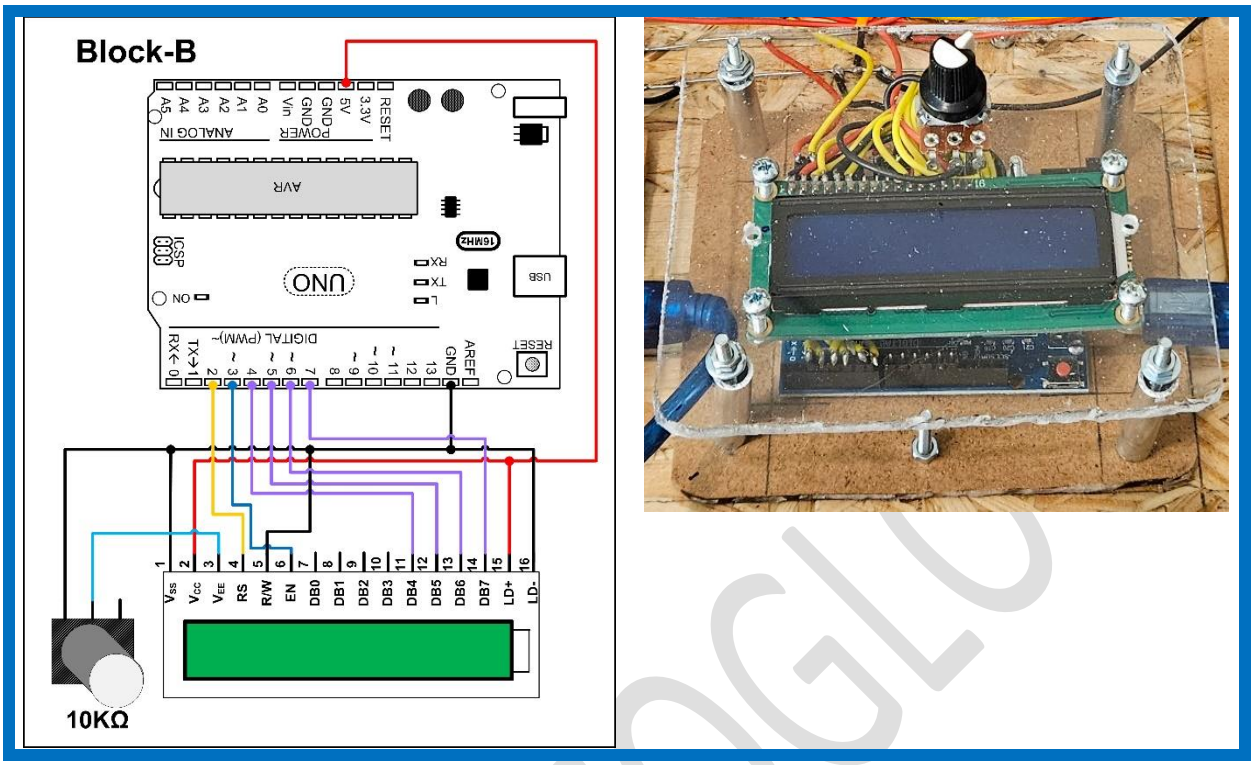


Fig. 2.4 (a) Block-B circuit and (b) physical implementation, code file: HOMS-ALU-SR.ino

On the other hand, block-C (fig. 2.5) has also two illuminated buttons and thus, more than two cables are needed for the corresponding connection. Each button has four pins in total. Tow pins are for the button operation, and two pins are for the embedded LED. The button operation is supported through the pull-up internal resistors of the microcontroller that are activated within the code. The button LEDs can be connected directly to 5V (using a resistor of 150Ω) for permanent illumination or in digital pins of Arduino for controlling illumination.

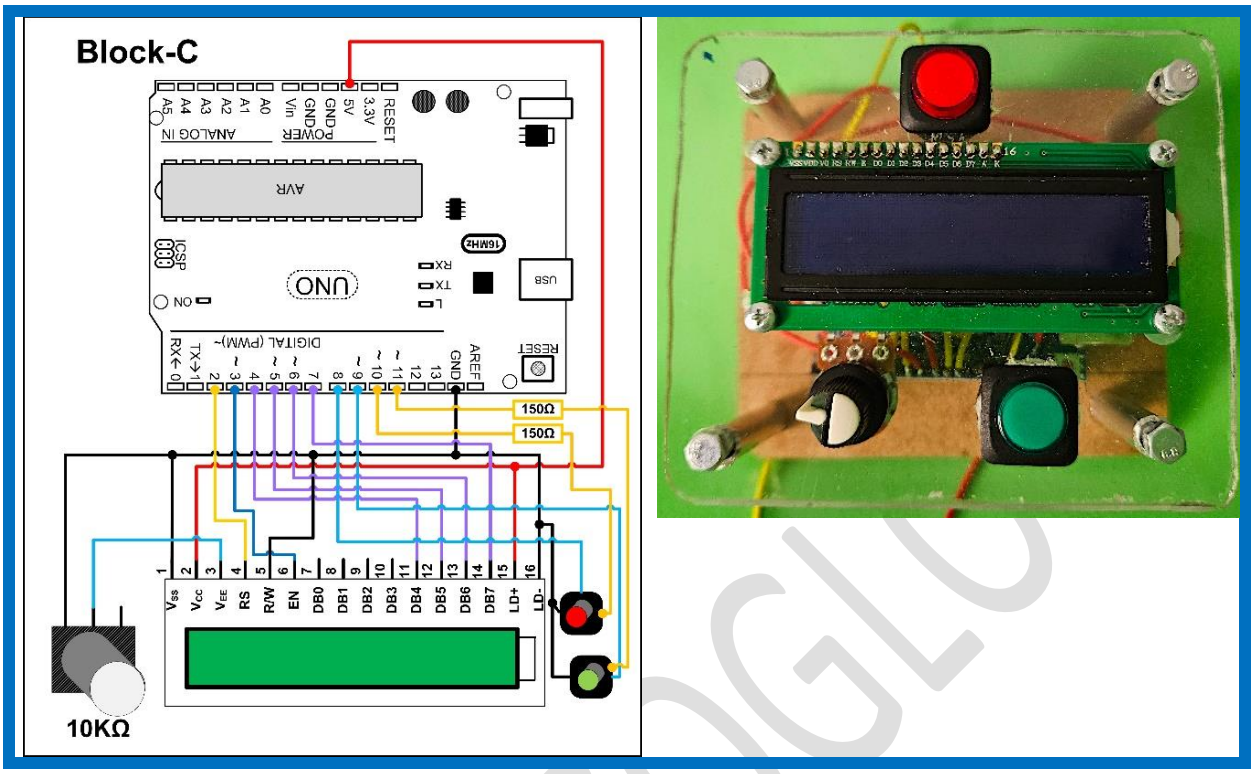


Fig. 2.5 (a) Block-C circuit and (b) physical implementation, code file: HOMS-CU.ino

Figure 2.6 shows the block-D circuit and implementation. Block-D has eight (8) switches and three (3) buttons.

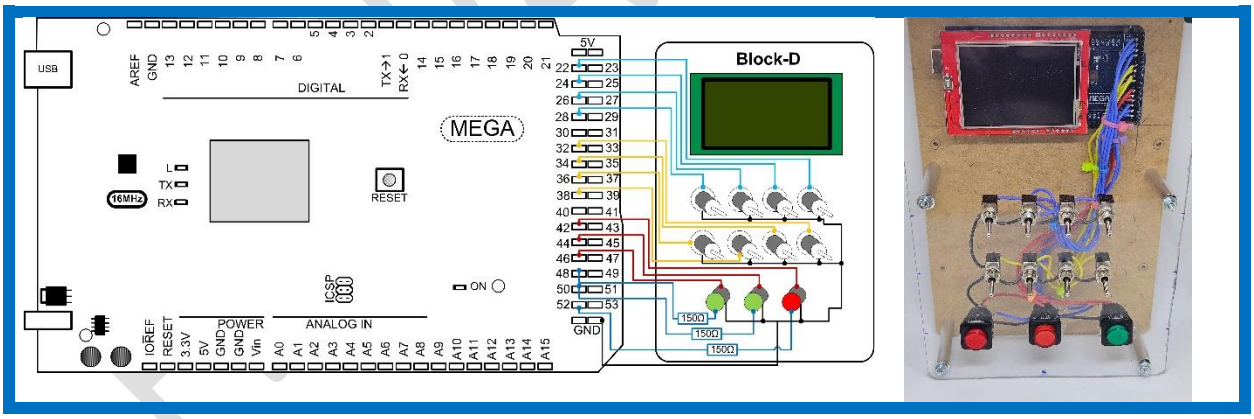


Fig. 2.6 (a) Block-D circuit and (b) physical implementation, code file: HOMS-MEM-OUT.ino

All the needed block circuits are very easy to be implemented. The concept of the HOMS tool is to combine the same simple blocks for building a fully working system. The whole system supports all the needed functions only if cooperation between blocks is performed.

2.4 Embedded software

***Important note:** only indicative instructions and operations have been implemented in the following source-codes for supporting basic system functionality.*

2.4.1 Register Unit

Every register Unit (Block A type, fig. 2.7) contains identical code for supporting instructions (operations) such as LOAD, READ, INC, DEC, SHIFT and RESET. Three basic functions (fig. 2.8) support the I^2C communication (onReceive, onRequest) and seven-segment display operation (display).

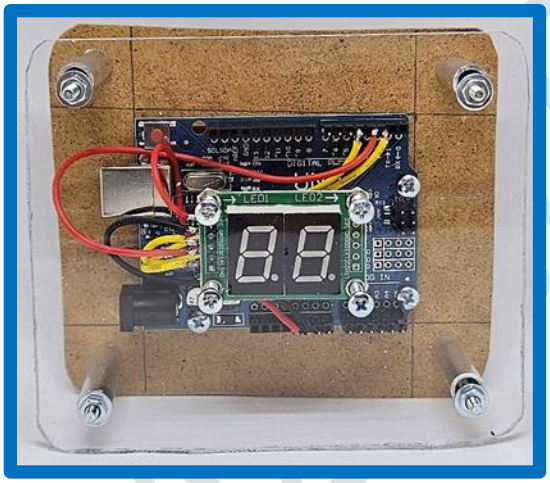


Fig. 2.7 Register Unit

When a single byte is received for executing an instruction regarding the register content, the function onReceive is activated. On the other hand, when the control unit requests an answer (e.g. READ register content), the function onRequest is activated. Finally, the function display is activated when the seven-segment display value has to be updated.

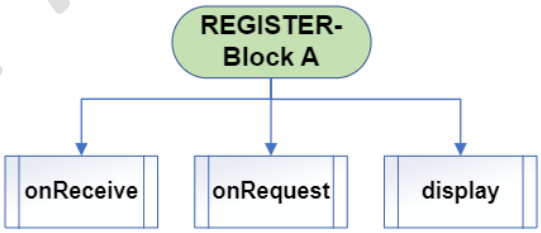


Fig. 2.8 Register functions

Figure 2.9 shows the operation of the function onReceive. In the case of the LOAD instruction, two bytes are used: one for the LOAD instruction (previous byte) and one for the value (current byte) that will be stored in the register. The function checks always the instruction code value for executing the corresponding operation (fig. 2.9).

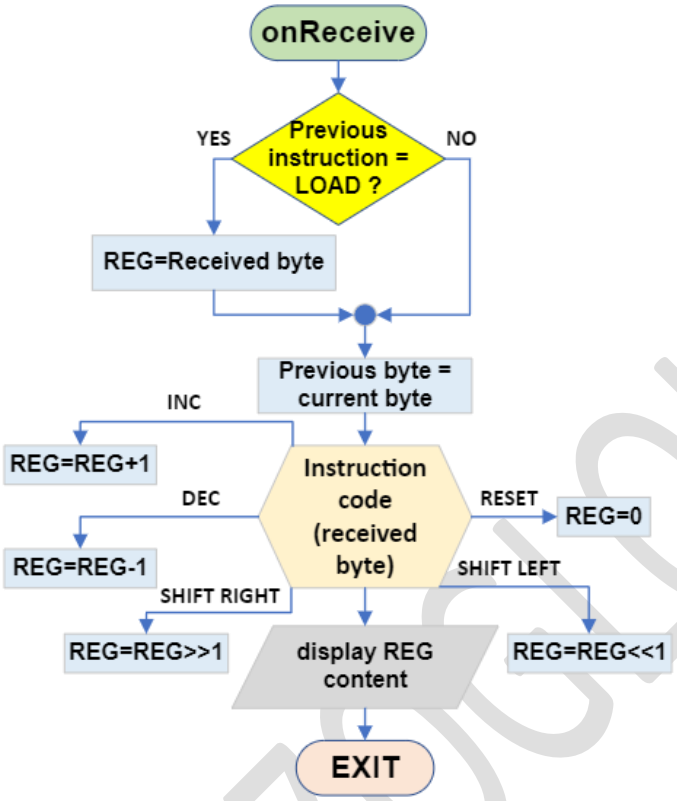


Fig. 2.9 onReceive function operation

Figure 2.10 shows the operation of the function onRequest.

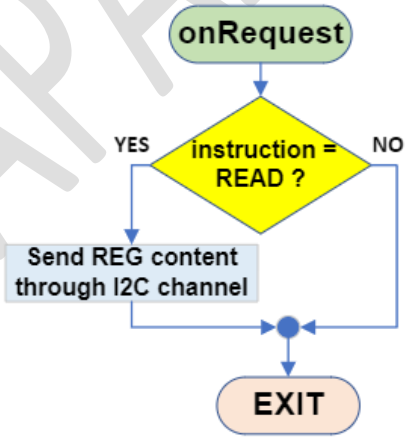


Fig. 2.10 onRequest function operation

SOURCE-CODE:

```
/*****************************************
REGISTER UNIT
All registers execute the same code,
the only difference is the I2C address
*****************************************
HOMS
Hardware-Oriented Microprocessor
Simulator
(C) Panayotis (Panos) Papazoglou
LICENSE:
Creative Commons
CC BY NC SA
International License
*****************************************/
//Change address based on block that is used
#define ADDRESS 2

//Include I2C library
#include <Wire.h>


/*****************************************
Declare PINS for seven segment display module
*****************************************/
#define Data 2
#define Clock 3
#define Load 4

//Define symbolic names for instructions codes
//between blocks
#define READ  65
#define RESET 66
#define SHIFT_L 67
#define SHIFT_R 68
#define DEC 69
#define INC 70
#define LOAD 71

/*****************************************
Segment status ON/OFF for one digit [0, F]
*****************************************/
byte hex[16]=
```

```
{
  0b00000011,  //0
  0b10011111,  //1
  0b00100101,  //2
  0b00001101,  //3
  0b10011001,  //4
  0b01001001,  //5
  0b01000001,  //6
  0b00011111,  //7
  0b00000001,  //8
  0b00001001,  //9
  0b00010001,  //a
  0b11000001,  //b
  0b11100101,  //c
  0b10000101,  //d
  0b01100001,  //e
  0b01110001   //f
};

//byte that is received via I2C communication
byte rV;

//Previous received byte. Is used when a sequence of bytes
//has to be checked
byte rVprev=0;

//Initial register content
byte REG=0x00;

//Starting function (after reset or power on), runs once
void setup() {

  //Initialize Serial communication
  //for displaying debugging messages
  Serial.begin(9600);

  //Initialize seven segment module
  init_display();

  //Display register content
  display(REG);

  //Initialize I2C communication:
  //Set address
  //Enable ISR for receiving bytes and requests
```

```
  Wire.begin(ADDRESS);
  Wire.onReceive(onReceive);
  Wire.onRequest(onRequest);
}

void loop()
{
  //Nothing here

}

/*********************************************
Action Routine that is activated when
a byte is received
*********************************************/
void onReceive(int a)

{

  //Read received byte
  rV=Wire.read();

  //If previous byte is 71 (LOAD), then the current byte
  //is the value that will be loaded in register
  if (rVprev==LOAD) REG=rV;

  //Update variable for previous value
  rVprev=rV;

  //Actions based on received byte
  if (rV==INC) REG++;
  if (rV==DEC) REG--;
  if (rV==SHIFT_R) REG=REG>>1;
  if (rV==SHIFT_L) REG=REG<<1;
  if (rV==RESET) REG=0;

  //Display register content after performed action
   display(REG);

}

/*********************************************
Action Routine that is activated when
a request is received
*********************************************/
```

```
void onRequest()
{

  //If the received byte represents
  //the READ command, then the
  //register content is sent as answer
  if (rV==READ) Wire.write(REG);

  //Display register content
  display(REG);
}

/*****************************************
Display register content as HEX number on
seven segment module
Digit manipulation for left and right
seven segment display unit
*****************************************/
void display(byte REGnum)
{

//Convert decimal number REGnum to HEX
String Shex=String(REGnum,HEX);

//Variables for left and right digit
char LL;
char RR;

//If the hex number has only one digit, then
//the left digit is zero ('0') and the right digit
//is the first character of the string Shex
if (Shex.length()==1)
  { LL='0'; RR=Shex[0];}

//Otherwise, update left and right digit variables from
//the whole string Shex
else
{
  LL=Shex[0];
  RR=Shex[1];
}

//variables for accessing later the hex[] array
//in order to activate the correct segment
//on seven segment module
```

```
byte left,right;

//Serial.print("LL="); Serial.println(LL);
//Serial.print("RR="); Serial.println(RR);

//Translate one digit numbers from string form
//to single numbers for accessing hex[] array
//The calculations are based on the corresponding ASCII
//code for each string digit
//for example, character 'a' has the ASCII code 97
//the calculation is 'a'-87=97-87=10. The number 10
//will be used as index in the hex[] array (hex[10])
//for displaying 'A'
if (LL>='a' && LL<='f') left=LL-87; else
   if (LL>='0' && LL<='9') left=LL-48;

  if (RR>='a' && RR<='f') right=RR-87; else
     if (RR>='0' && RR<='9') right=RR-48;


//Serial.print(left); Serial.print(", "); Serial.println(right);

  //Display final hex number to seven segment display module
  digitalWrite(Load,LOW);
  shiftOut(Data,Clock, LSBFIRST, hex[right]);
  shiftOut(Data,Clock, LSBFIRST, hex[left]);
  digitalWrite(Load,HIGH);

}


/*****************************************
Initialize
seven segment module
*****************************************/
void init_display()
{
  pinMode(Data,OUTPUT);
  pinMode(Clock,OUTPUT);
  pinMode(Load, OUTPUT);
}
```

2.4.2 Arithmetic & Logic Unit (ALU) and Status Register (SR)

In current code version, only the ADD instruction is implemented. When an ADD instruction is received, the content of temporary registers (input buffers) inside the ALU unit is read, and then the addition T1+T1 is performed.

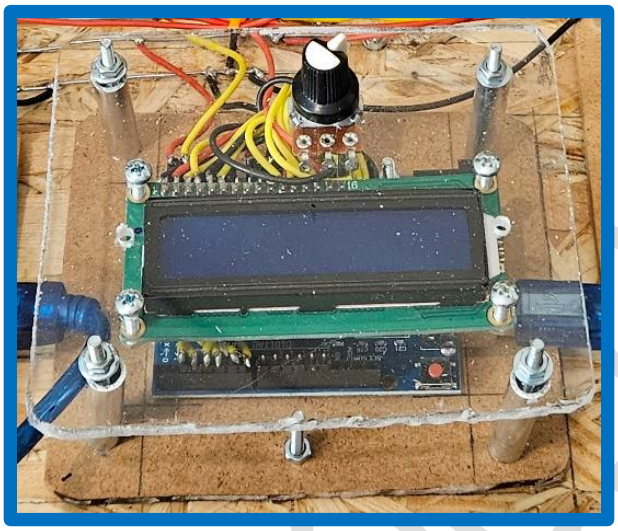


Fig 2.11 ALU-SR unit

The result is displayed on the LCD screen (fig 2.11) and the Status Register is updated. The result is available inside the CPU model (simulator) through the internal data bus.

The following pseudo-code represents the above steps for the main function operation:

START

Byte received?

YES

* Instruction=ADD

* YES

* * Read Temporary register T1

* * Read Temporary register T2

* * Result=T1+T2

* * Update Status Register (SR)

* * Display REG content

* NO

* -

NO

-

END

Function list

onReceive
when a byte is received (no answer is requested)

onRequest
when a byte is received and an answer is requested

READ_REG
read the content of another register (by setting the corresponding I^2C address)

UPDATE_SR
update the status register (SR) content

DISPLAY_RES
display result and SR content on the LCD screen

The status register (SR) is declared as a three element array (*byte SR[]={0,0,0};*)

SOURCE-CODE:

```
/*****************************************
ALU (Arithmetic and Logic Unit)
SR (Status Register)
UNIT
*****************************************
HOMS
Hardware-Oriented Microprocessor
Simulator
(C) Panayotis (Panos) Papazoglou
LICENSE:
Creative Commons
CC BY NC SA
International License
*****************************************/
//UNIT ADDRESS = 9
#define ADDRESS 9

//Include I2C library
#include <Wire.h>

//Include LCD library
#include <LiquidCrystal.h>

//Set pins for LCD1602 connection
LiquidCrystal lcd(2,3,4,5,6,7);

//This UNIT Receives/Sends commands from/to other units as bytes
#define READ  65
#define ADD 72

//Variable that is used inside ISR
volatile byte rvTRUE=0;

//Byte received from other units
byte rV;

//Store result of the numerical calculation
byte result=0;

//Store the content of temporarily registers T1 and T2
byte T1=0;
byte T2=0;
```

```
//Flag array
byte SR[]={0,0,0}; //VF (oVerflow), ZF (Zero), SF (Sign)

//Starting function (after reset or power on), runs once
void setup()
{
  //Start I2C communication
  Wire.begin(ADDRESS);

  //Activate Receive & Request ISR routines
  Wire.onReceive(onReceive);
  Wire.onRequest(onRequest);

  //Initialize LCD screen
  lcd.begin(16,2);

  //Display result and flag status
  DISPLAY_RES();
}

void loop()
{
  //If a byte is received and is an ADD command,
  //then execute addition between the contents
  //of special registers T1 and T2
  //Store result in a variable, update SR[] array
  //and display result/SR contents
  if (rvTRUE==1)
  {
     if (rV==ADD)
      {
         T1=READ_REG(7);
         delay(100);
         T2=READ_REG(8);
         result=T1+T2;
         UPDATE_SR(result);
         DISPLAY_RES();
      }
     rvTRUE=0;
  }
}
```

```
/*****************************************
ISR routine
Is activated when a byte is received
via I2C communication
*****************************************/
void onReceive(int a)
{
  rV=Wire.read();

  //Flag for activating code in order to read T1 and T2 contents
  //I2C functions can not be called within an active I2C routine
  rvTRUE=1;
}

/*****************************************
ISR routine
Is activated when a request is received
via I2C communication
*****************************************/
void onRequest()
{
  //Send result (ALU calculation) as answer to an I2C request
  Wire.write(result);
}

/*****************************************
Read the content of a Register
with address addr
*****************************************/
byte READ_REG(byte addr)
{
byte ans;

  Wire.beginTransmission(addr);
  Wire.write(READ);
  Wire.endTransmission();

  //Receive answer
  Wire.requestFrom(addr, 1);
  if (Wire.available())
    ans = Wire.read();

return ans;
}
```

```
/*****************************************
Update Status Register Flags based
on result status
*****************************************/
void UPDATE_SR(byte res)
{

if (res==0) SR[1]=1; else SR[1]=0;
if (res<0) SR[2]=1; else SR[2]=0;
if (res>255) {SR[0]=1; result=0;} else SR[0]=0;
}

/*****************************************
Display result and Status Register Flags
on LCD screen
*****************************************/
void DISPLAY_RES()
{
  lcd.setCursor(0,0); lcd.print("RES10="); lcd.setCursor(6,0);
  lcd.print(result);
  lcd.setCursor(10,0); lcd.print("HEX="); lcd.setCursor(14,0);
  lcd.print(String(result,HEX));
  lcd.setCursor(4,1); lcd.print("V:"); lcd.setCursor(6,1); lcd.print(SR[0]);
  lcd.setCursor(8,1); lcd.print("Z:"); lcd.setCursor(10,1); lcd.print(SR[1]);
  lcd.setCursor(12,1); lcd.print("S:"); lcd.setCursor(14,1); lcd.print(SR[2]);
}
```

2.4.3 Memory and I/O system

The HOMS system executes the assembly instructions that are stored inside the memory. Thus, a memory system is implemented. Additionally, the HOMS supports user input for entering assembly instructions in memory. For instruction data entry, a hardware user interface is also developed.

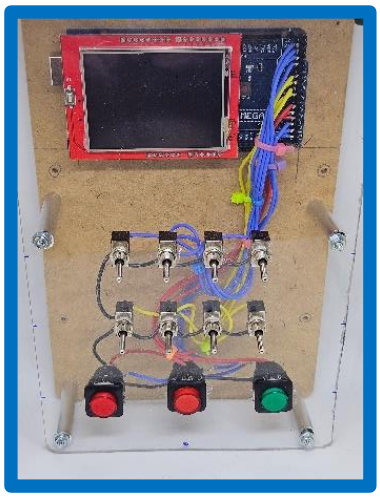


Fig. 2.12 MEM, I/O unit

Function list

init_sw_buttons
Initialize switches and buttons using microcontroller PULL_UP resistors

init_buttonLEDs
Initialize button LEDs

init_TFT
Initialize TFT screen

TFT_welcome
Display welcome messages

print_dechex
Display DEC and HEX value on TFT screen based on 8 states of the switches

print_text
display text (String) on the TFT screen

clear_TFT
clear TFT and fill with color

init_mem
initialize memory (integer array of 256 locations)

onReceive
receive from control unit (e.g. LOAD instruction for storing data in memory)

onRequest
control unit requests the content of a given memory address

set_prog
for auto-filling memory locations from prog[] array (preloaded instructions for running the first demo)

SOURCE-CODE:

```
/*****************************************
MEMORY & OUTPUT UNIT
*****************************************
HOMS
Hardware-Oriented Microprocessor
Simulator
(C) Panayotis (Panos) Papazoglou
LICENSE:
Creative Commons
CC BY NC 3.0
International License
*****************************************/

/*****************************************
LIBRARIES for the TFT 2.4 Inch screen
*****************************************/
#include <Adafruit_GFX.h>
#include <MCUFRIEND_kbv.h>

//Include I2C library
#include <Wire.h>
//Object declaration for accessing TFT library functions
//Every function call starts with the prefix tft.
MCUFRIEND_kbv tft;

//TFT Color codes for easy access
#define BLACK   0x0000
#define BLUE    0x001F
#define RED     0xF800
```

```
#define GREEN   0x07E0
#define CYAN    0x07FF
#define MAGENTA 0xF81F
#define YELLOW  0xFFE0
#define WHITE   0xFFFF

//A byte is read from memory when the LOAD command is received
#define LOAD 71

//8bit switch pins
int bitsw[]={28,26,24,22,38,36,34,32};

//Button pins
int button[]={42,44,46};

//Button LEDs pins
int buttonLED[]={48,50,52};

//Button state
int bstate[]={0,0,0};

//Switch state 1=inactive (due to PULL_UP resistors)
int swstate[]={1,1,1,1,1,1,1,1};

//Memory: 256 locations, starting address 0, ending address 255 (FF hex)
int mem[256];

//Current selected address
int addr=0;

//Current data for storing in memory
int data=0;

//Keep previous values for updating screen only if values are changed
int addrprev=0, dataprev=0;

//Number in dec format
float dec=0;

//Integer dec number
int idec=0;

//Previous dec value (round is used before decprev update)
int decprev=0;
```

```
//Previous hex value (round is used before hexprev update)
String fhexprev;

//Received data from control unit
byte rV=0;

//Previous received data from control unit
byte rVprev=0;

//Flag that is activated when an address is received
byte ADDRflag=0;

//Declare next array if you want to have a preloaded code in memory
byte prog[]={4, 8, 10, 0, 6, 0, 14, 0, 17, 0};

//Starting function (after reset or power on), runs once
void setup(void)
{

  //initialize I2C communication (address 11)
  Wire.begin(11);

  //Enable ISR routines for receiving bytes and requests
  Wire.onReceive(onReceive);
  Wire.onRequest(onRequest);

  //Initialize Serial communication
  //for displaying debugging messages
  Serial.begin(9600);

  //initialize switches and buttons
  init_sw_buttons();

  //initialize button LEDs
  init_buttonLEDs();

  //initialize memory
  init_mem(256);

  //Store prog[] bytes in memory (preload code)
  set_prog();

  //initialize TFT
  init_TFT();
```

```
  //display welcome messages on TFT
  TFT_welcome();
}


/*****************************************
Main function (always active)
*****************************************/
void loop(void)
{
/*****************************************
Read switch states (8 switches)
store each state in swstate[] array
use bitsw[] array for switch pins
*****************************************/
  for(int i=0;i<8;i++)
    swstate[i]=digitalRead(bitsw[i]);

  //print dec and hex value on screen (based on switches state)
  print_dechex();

/*****************************************
Read button states (3 buttons)
store each state in bstate[] array
use button[] array for button pins
*****************************************/
for(int i=0;i<3;i++)
    bstate[i]=digitalRead(button[i]);

/*********************************************************************************
Check button states from bstate[] array
pins:
button[0]=GREEN, button[1]=RED RIGHT, button[2]=RED LEFT
if the RED LEFT button is pressed, the dec value is stored as address
if the RED RIGHT button is pressed, the dec value is tored as data (content)
if the GREEN button is pressed, the user wants to store data in the address addr
or wants to display memory contents on the screen
*********************************************************************************/
if (bstate[2]==LOW) addr=round(dec); if (bstate[1]==LOW) data=round(dec);
if (bstate[0]==LOW)
{
  //If both addr and data values are 128, the memory contents are displayed on the TFT screen
  if ((addr==128) && (data==128))
  {
    //Clear TFT screen
```

```
  clear_TFT(BLACK);

  //Display titles for the two DATA columns
  print_text(YELLOW, RED,2,10,10,"ADDR"); print_text(YELLOW, RED,2,60,10,"DATA");
  print_text(YELLOW, RED,2,110,10,"ADDR"); print_text(YELLOW, RED,2,160,10,"DATA");

  //Variables for moving to the desired X,Y position
  int x=10, y=25;

  //Loop for display memory contents, 1st column from 0 to 10, 2nd column from 11 to 21
  for(int i=0;i<11;i++)
  {
    //Display address for first range
    print_text(WHITE, BLACK,2,x,y,String(i));

    //Display address content for first range
    print_text(WHITE, BLACK,2,x+50,y,String(mem[i]));

    //Display address for second range
    print_text(WHITE, BLACK,2,x+100,y,String(i+11));

    //Display address content for second range
    print_text(WHITE, BLACK,2,x+150,y,String(mem[i+11]));

    //Update Y position for changing line
    y+=20;

  }

  //Display EXIT label
  print_text(WHITE, BLUE,2,220,180,"EXIT");

  //Wait on that screen until green button is pressed
  while(digitalRead(button[0])==HIGH) {;}

  //Clear TFT screen
  clear_TFT(BLACK);

}
else
//the user has selected to store data value in memory
{

//Display SET for confirming green button press
print_text(YELLOW, RED,3,220,180,"SET");
```

```
  //Store data value in memory (mem[] array)
  mem[addr]=data;

  delay(100);

  //Display OK. The store process is successful
  print_text(WHITE, BLUE,3,220,180,"OK!");
  delay(500);
  }
}

//If a new ADDRESS is entered, clear for the new value
if (addr!=addrprev) tft.fillRect(50, 180,100,35,BLACK);

//If new DATA are entered, clear for the new value
if (data!=dataprev) tft.fillRect(150, 180,100,35,BLACK);

//Store current values of data and address before the next update
dataprev=data; addrprev=addr;

//Display current address value (updated with the RED LEFT button based on switches state)
print_text(WHITE, RED,3,10,180,"A:");
print_text(YELLOW, RED,3,50,180,String(addr));

//Display current data value (updated with the RED RIGHT button based on switches state)
print_text(WHITE, RED,3,120,180,"D:");
print_text(YELLOW, RED,3,150,180,String(data));

//Display label for GREEN button
print_text(BLACK, GREEN,3,220,180,"SET");

//Delay before return to loop starting point
delay(30);
}

/*********************************************
Initialize switches and buttons
using microcontroller PULL_UP resistors.
If a switch or a button is inactive
the corresponding state is HIGH
*********************************************/
void init_sw_buttons()
{
  for(int i=0;i<8;i++)
```

```
    pinMode(bitsw[i],INPUT_PULLUP);

  for(int i=0;i<3;i++)
    pinMode(button[i],INPUT_PULLUP);
}


/*****************************************
Initialize button LEDs.
Set all pins to HIGH.
Lighting the LEDs
*****************************************/
void init_buttonLEDs()
{
  for(int i=0;i<3;i++)
  {
    pinMode(buttonLED[i],OUTPUT);
    digitalWrite(buttonLED[i],HIGH);
  }

}



/*****************************************
Initialize TFT screen
rotate coordinate system to use
screen as landscape
*****************************************/
void init_TFT()
{
   uint16_t ID = tft.readID();
    tft.begin(ID);
    tft.setRotation(1);            //PORTRAIT
    clear_TFT(BLACK);
}

/*****************************************
Display welcome messages
*****************************************/
void TFT_welcome()
{
  clear_TFT(BLUE);
  int x=30;
  print_text(WHITE,BLUE, 3, x,30,"Hardware");
```

```
  print_text(WHITE,BLUE, 3, x,60,"based");
  print_text(WHITE,BLUE, 3, x,90,"Microprocessor");
  print_text(WHITE,BLUE, 3, x,120,"Simulator V1.0");
  print_text(YELLOW,BLUE, 2, 20,160,"(C) P. PAPAZOGLOU, 2023");
  print_text(WHITE, BLUE,2,60,200,"-- PRESS GREEN --");

  //Wait on that screen until green button is pressed
  while(digitalRead(button[0])==HIGH) {;}

  clear_TFT(BLACK);

   }

/*****************************************
Display DEC and HEX value on screen
based on 8 states of the switches
8 states = 8 bits
*****************************************/
void print_dechex()
{
  //Initialize dec value
  dec=0;

  //Exponent of two starting from the MSB bit
  int p=7;

  //Variable for bit value based on switches state (switch=0=active, bit=1)
  int b;

  //Variables for moving to the desired X,Y locations
  int x=10, y=50;

  //Display label for the binary representation
  print_text(WHITE, BLACK, 5, 10, 10, "BIN");

  //Display 8bit value and calculate the decimal value
  for(int i=0;i<8;i++)
  {
    //Set bit value based on switch state
    if (swstate[i]==LOW) b=1; else b=0;

    //Display current bit
    print_text(YELLOW, BLACK, 4, x, y, String(b));

    //Update X location for displaying the next right bit
```

```
  x+=30;

  //Calculate dec value for current bit (b*2^p)
  float value=b*pow(2,p);

  //Update sum for the whole number
  dec=dec+value;

  //Decrease the exponent (based on bit position in the number)
  p--;
 }

 //idec is the integer representation of dec (round function is used
 //for value integrity due to type conversion)
 idec=round(dec);

 //Convert decimal value to hexadecimal (hex)
 String fhex =  String(idec, HEX);

 //For a new hex value, clear display field
 if (fhex.compareTo(fhexprev)!=0) tft.fillRect(180, 120,50,40,BLACK);

 //Keep current hex value (as previous)
 fhexprev=fhex;
 //Display label and hex value on screen
 print_text(WHITE, BLACK, 4, 150, 90, "HEX");
 print_text(YELLOW, BLACK, 4, 170, 130, fhex);

 //For a new dec value, clear display field
 if (idec!=decprev) tft.fillRect(10, 120,215,40,BLACK);

 //Keep current dec value (as previous)
 decprev=idec;

 //Display label and dec value on screen
 print_text(WHITE, BLACK, 5, 10, 90, "DEC");
 print_text(YELLOW, BLACK, 4, 10, 130, String(idec));
}
/*****************************************
display text (String) on the TFT screen

fcolor: text color
bcolor: text background color
text_size: size of text
xpos: X position on screen
```

```
ypos: Y position on screen
text: text to be displayed
*****************************************/
void print_text(int fcolor, int bcolor, int text_size, int xpos, int ypos, String text)
{
  tft.setTextColor(fcolor, bcolor);
    tft.setTextSize(text_size);
    tft.setCursor(xpos, ypos);
    tft.print(text);
}
/*****************************************
clear TFT screen, fill with color
*****************************************/
void clear_TFT(int color)
{
tft.fillScreen(color);

}
/*****************************************
initialize memory with size of locations

locations: total array locations
*****************************************/
void init_mem(int locations)
{
  for(int i=0;i<locations;i++)
    mem[i]=0;
}

/*****************************************
When data are received from control unit
*****************************************/
void onReceive(int a)
{
  rV=Wire.read();
  Serial.print("rV:"); Serial.println(rV);

  //If previous command is LOAD, then the current byte represents an address
  if (rVprev==LOAD) ADDRflag=1;
  rVprev=rV;
}
```

```
/*****************************************
When data are requested from control unit
*****************************************/
void onRequest()
{
   //If the contents of an address are requested
   if (ADDRflag==1)
   {
     Serial.println("ADDRflag=1");
     Serial.print("rV:");Serial.print(rV);
     Serial.print(", content:"); Serial.println(mem[rV]);
     Wire.write(mem[rV]);
     ADDRflag=0;
     rVprev=0;
   }
}
/*****************************************
Preload program bytes in memory.
Update mem[] array from prog[] array
*****************************************/
void set_prog()
{
  for(int i=0;i<10;i++) mem[i]=prog[i];
}
```

2.4.4 Control Unit (CU)

The control unit constitutes the most important component of the HOMS system. This unit fetches instruction data from the memory unit, decodes each instruction (type and parameters) and finally sends commands to other components for supporting the executing cycle.

In other words, synchronizes the HOMS components operation for supporting a fully working system regarding the instruction execution. The required steps for each instruction execution are implemented within the CU.

For creating new assembly instructions, new source code has to be added inside the CU. The CU source-code is the longest code as compared to other HOMS units.

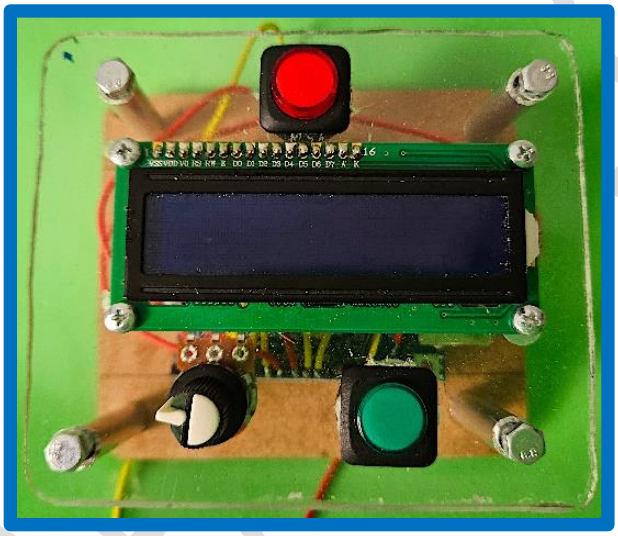


Fig. 2.13 CU unit

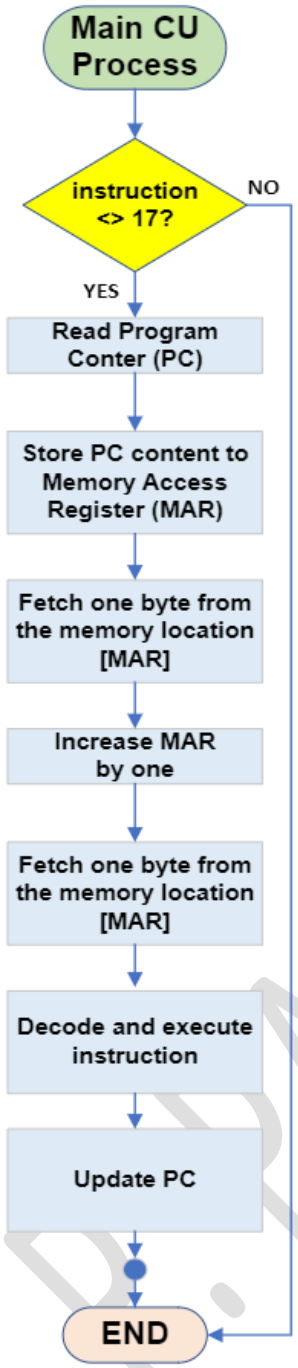


Fig. 2.14 Execution process

The flow chart (fig 2.14) shows the whole procedure for (a) fetching instruction data from memory and (b) executing instruction through the control unit. This procedure can be described in steps as follows:

Step 1: Read PC (Program counter) for finding where the next instruction address is

Step 2: Store this address to MAR (Memory Access Register) in order to place the desired memory address on address bus

Step 3: Fetch instruction code via data bus and store this code in MBR (Memory Buffer register) and increase MAR by one

Step 4: Fetch parameter code via data bus and store this code in MBR (Memory Buffer register)

Step 5: Decode instruction data and execute, a) copy the contents of registers A and B to ALU registers T1 and T2 respectively, b) perform the addition T1+T2 and update SR (Status Register), c) store result to register A

Step 6: Update PC (Program Counter) for fetching next instruction code from memory

SOURCE-CODE:

```
/*****************************************
CONTROL UNIT
I2C MASTER
*****************************************
HOMS
Hardware-based Microprocessor
Simulator
(C) Panayotis (Panos) Papazoglou
LICENSE:
Creative Commons
CC BY NC SA
International License
*****************************************/
//Please ignore any compilation warnings
//due to function overload (the same function name
//can be used but with different parameters).
//the MCU executes the function which matches
//with parameters

//Define symbolic names for sending commands to other blocks
//The I2C communication is based on one byte transmit/receive
#define READ  65
#define RESET 66
#define SHIFT_L 67
#define SHIFT_R 68
#define DEC 69
#define INC 70
#define LOAD 71
#define ADD 72

//Set symbolic names for button/LED pins
#define GREEN_button 11
#define RED_button 12
#define GREEN_LED 10
#define RED_LED 9

//Include I2C library
//The I2C function are called using the prefix Wire.
#include <Wire.h>

//Include LCD library
#include <LiquidCrystal.h>
```

```
//Set pins for LCD1602 connection
LiquidCrystal lcd(3,4,5,6,7,8);

//Declare variables for the needs of the CU unit (locally)
byte RA=0;
byte RB=0;
byte RC=0;
byte PC=0;
byte MAR=0;
byte MBR=0;
byte T1=0;
byte T2=0;
byte RS=0;
byte command=0;
byte param=0;
String hexval="";
byte blockADDR;

//Delay between execution process steps
int d=3000;

//Starting function (after reset or power on), runs once
void setup()
{

  //Initialize I2C communication
  Wire.begin();

  //Initialize Serial communication
  //for displaying debugging messages
  Serial.begin(9600);

  //Initialize buttons & LED-buttons
  init_buttons_led();
  //Initialize LCD
  lcd.begin(16, 2);

  //Initial value for entering the while loop
  command=0;

  //Clear registers (content=00)
  //Wait for RED button to start execution
  INIT();
```

```
  /*****************************
  MAIN EXECUTION PROCESS
  *****************************/
  while (command!=17)
  {
  READ_PC();    //Read PC content for instruction starting address
  PC2MAR();     //Update MAR for accessibg memory address
  FETCH1();     //Fetch 1st byte from memory (instruction code)
  UPDATE_MAR(); //Update MAR for fetching 2nd byte
  FETCH2();     //Fetch 2nd byte (instruction parameter)
  DECODE_EXEC();//Decode instruction and execute
  UPDATE_PC();  //Update PC content for next instruction

  //Uncomment the following two code lines if you want a pause
  //between instruction execution

  //lcdtext(0,1,"Press <NEXT>");
  //wait();
  }

  //Out of while loop. The following code is executed when the instruction code is 17
  //which corresponds to STOP instruction
  lcd.clear();
  lcdtext(1,0,"END OF PROGRAM");
  lcdtext(3,1,"EXECUTION");

}//End of SETUP


/*****************************************
READ PC (Program Counter), PC block address=4
*****************************************/

void READ_PC()
{
  //Display message on LCD
  lcd.clear();
  lcdtext(0,0,"Reading PC REG..");
  delay(d);

  //Read PC content. Store content in PC variable
  PC=READ_REG(4);

  //Display READ results on LCD
  lcd.clear();
```

```
  lcd.setCursor(0,0); lcd.print("PC-Prog.Counter");
  lcd.setCursor(0,1); lcd.print("PC10=");
  lcd.setCursor(5,1); lcd.print(PC);
  lcd.setCursor(9,1); lcd.print("PC16=");
  hexval =  String(PC, HEX);
  lcd.setCursor(14,1); lcd.print(hexval);

  //Wait before next step
  delay(d);

} //End of READ_PC

/*****************************************
Store PC contents to register MAR
for accessing memory address
MAR=PC (MAR block address=5)
*****************************************/
void PC2MAR()
{
  //Display message on LCD
  lcd.clear();
  lcdtext(0,0,"MAR <-- PC ...");
  delay(d);


  //LOAD PC content in MAR register
  WRITE_REG(5,PC);

  //Update variable
  MAR=PC;

  //Display message on LCD
  lcd.clear();
  lcdtext(0,0,"Check MAR...");
  delay(d);
}//End of PC2MAR

/*****************************************
FETCH 1st byte from memory, mem[MAR]
Instruction code
*****************************************/
void FETCH1()
{
  //Display message on LCD
  lcd.clear();
```

```
  lcdtext(0,0,"FETCH byte 1...");
  delay(d);

  //LOAD MAR content in MEM/OUT Unit
  //for accessing the corresponding address
  WRITE_REG(11,MAR);

  //Receive memory content
  //Store content in MBR variable
  blockADDR=11;
  Wire.requestFrom(blockADDR, 1);
  if (Wire.available())
    MBR = Wire.read();

//Store instruction code in command variable
command=MBR;

//Display information (MAR, MBR) on LCD
lcd.clear();
lcd.setCursor(0,0); lcd.print("MBR<--mem[");
lcd.setCursor(10,0); lcd.print(MAR);
lcd.setCursor(13,0); lcd.print("]");
lcd.setCursor(0,1); lcd.print("(10)=");
lcd.setCursor(5,1); lcd.print(MBR);
lcd.setCursor(9,1); lcd.print("(16)=");
hexval =  String(MBR, HEX);
lcd.setCursor(14,1); lcd.print(hexval);
delay(d);

//Update MBR register from MBR variable
WRITE_REG(6,MBR);

//Display message on LCD
lcd.clear();
lcdtext(0,0,"Check MBR");
delay(d);
lcd.clear();
lcdtext(0,0,"command is"); lcd.setCursor(11,0); lcd.print(command);
delay(d);
} //End of FETCH1


/*****************************************
Updating MAR for accessing
MAR=PC+1 (MAR address=5)
```

```
*****************************************/
void UPDATE_MAR()
{
  //Display message on LCD
  lcd.clear();
  lcdtext(0,0,"Updating MAR...");

  //Update MAR register for fetching the next byte from memory
  //(from next address)
  WRITE_REG(5,PC+1);
  delay(d);

  //Update variable
  MAR=PC+1;

  //Display message on LCD
  lcd.clear();
  lcdtext(0,0,"Check MAR");
  delay(d);
}


/*****************************************
FETCH 2nd byte from memory, mem[MAR]
Parameter code
*****************************************/
void FETCH2()
{
//Display message on LCD
lcd.clear();
lcdtext(0,0,"FETCH byte 2...");

//LOAD MAR content in MEM/OUT Unit
//for accessing the corresponding address
WRITE_REG(11,MAR);
delay(d);


//Receive memory content
//Store content in MBR variable
blockADDR=11;
Wire.requestFrom(blockADDR, 1);
if (Wire.available())
    MBR = Wire.read();
```

```
//Store parameter code in param variable
param=MBR;

//Display information (MAR, MBR) on LCD
lcd.clear();
lcd.setCursor(0,0); lcd.print("MBR<--mem[");
lcd.setCursor(10,0); lcd.print(MAR);
lcd.setCursor(13,0); lcd.print("]");
lcd.setCursor(0,1); lcd.print("(10)=");
lcd.setCursor(5,1); lcd.print(MBR);
lcd.setCursor(9,1); lcd.print("(16)=");
hexval =  String(MBR, HEX);
lcd.setCursor(14,1); lcd.print(hexval);
delay(d);

//update register MBR
WRITE_REG(6,MBR);

//Display messages on LCD
lcd.clear();
lcdtext(0,0,"Check MBR");
delay(d);
lcd.clear();
lcdtext(0,0,"param is"); lcd.setCursor(11,0); lcd.print(param);
delay(d);
} //END of FETCH2


/*****************************************
Decoding instruction and execution
Instruction block: [command][param]

Implemented Assembly Instructions

CODE  INSTRUCTION
====  ===========
04    LOD A,num
10    INC A
06    MOV B,A
14    ADD A,B
17    STOP

command=instruction code
param=parameter code
*****************************************/
```

```
//command,param
void DECODE_EXEC()
{

/*****************************
LOD A,num (A=num)
*****************************/
if (command==4)
{
  //Display messages on LCD
  lcd.clear();
  lcdtext(0,0,"CMD:LOD A,"); lcd.setCursor(10,0); lcd.print(param);
  lcdtext(0,1,"Executing...");
  delay(d);

  //LOAD param (num) in Register A
  WRITE_REG(1,param);

}

/*****************************
INC A (A=A+1)
*****************************/
if (command==10)
{
    //Display messages on LCD
    lcd.clear();
    lcdtext(0,0,"CMD:INC A");
    lcdtext(0,1,"Executing...");
    delay(d);

    //Send INC command to register A
    blockADDR=1;
    Wire.beginTransmission(blockADDR);
    Wire.write(INC);
    Wire.endTransmission();

    //Display messages on LCD
    lcd.clear();
    lcdtext(0,0,"Check REG-A");
}

/*****************************
MOV B,A (B=A)
*****************************/
```

```
if (command==6)
{
  //Display messages on LCD
  lcd.clear();
  lcdtext(0,0,"CMD:MOV B,A");
  lcdtext(0,1,"Executing...");
  delay(d);

  //Read from register A
  //Store content to variable RA
  RA=READ_REG(1);
  Serial.print("Read=RA:"); Serial.println(RA);

  //Store register A content (RA) in register B
  WRITE_REG(2,RA);

  //Display message on LCD
  lcd.clear();
  lcdtext(0,0,"Check REG-B");
}

/*****************************
ADD A,B
*****************************/
if (command==14)
{
  //Display message on LCD
  lcd.clear();
  lcdtext(0,0,"CMD:ADD A,B");
  delay(d);

//READ from register A, Store in variable RA
RA=READ_REG(1);

//READ from register B, Store in variable RB
RB=READ_REG(2);

//Store register A content in register T1
WRITE_REG(7,RA);

//Store register B content in register T2
WRITE_REG(8,RB);

//Display messages on LCD
lcd.clear();
```

```
lcdtext(0,0,"Check T1,T2");
lcdtext(0,1,"ALUEXEC...");
delay(d);

//Send ADD instruction to ALU/SR Unit
byte RES=SEND_ALU(9,ADD);

//Store ALU result in register A
WRITE_REG(1,RES);

//Display message on LCD
lcd.clear();
lcdtext(0,0,"Check RA,ALU,SR");
delay(d);
}


}//End of DECODE_EXEC

/*****************************************
Update PC register content
for fetching next instruction bytes
from memory
*****************************************/
void UPDATE_PC()
{

   //Send two INC commands to PC register
   Wire.beginTransmission(4);
   Wire.write(INC);
   Wire.endTransmission();
   delay(50);
   Wire.beginTransmission(4);
   Wire.write(INC);
   Wire.endTransmission();
   delay(50);

   //Update variable
   PC=PC+2;
}

void loop()
{
  //Nothing here
}
```

```
/*****************************************
Initialize PULL-UP resistors
Every button is activated when
a PULL-UP pin goes to LOW (GND)

button-state=1 inactive
button-state=0 active

Activate LEDs inside buttons
*****************************************/
void init_buttons_led()
{
  pinMode(GREEN_button,INPUT_PULLUP);
  pinMode(RED_button,INPUT_PULLUP);

  pinMode(GREEN_LED, OUTPUT);
  pinMode(RED_LED, OUTPUT);

  digitalWrite(GREEN_LED,HIGH);
  digitalWrite(RED_LED,HIGH);


}

/*****************************************
Pause
wait on GREEN button press
*****************************************/
void wait()
{
  while(digitalRead(GREEN_button)==HIGH) {;}
  delay(250);
}

/*****************************************
Pause
wait on RED button press
*****************************************/
void wait_red()
{
  while(digitalRead(RED_button)==HIGH) {;}
  delay(250);
```

```
}

/*****************************************
Function for displaying Strings on LCD
*****************************************/
void lcdtext(int col, int lin, String text)
{
  lcd.setCursor(col,lin);
  lcd.print(text);

}

/*****************************************
Clear all registers (set content=0)

Register address = 1 = Register A
Register address = 2 = Register B
Register address = 3 = Register C
Register address = 4 = Register PC
Register address = 5 = Register MAR
Register address = 6 = Register MBR
Register address = 7 = Register T1
Register address = 8 = Register T2
*****************************************/
void clear_REGS()
{
  for(int i=1;i<=8;i++)
  {
    Wire.beginTransmission(i);
    Wire.write(RESET);
    Wire.endTransmission();
    delay(150);
  }
}

/*****************************************
Display messages on LCD
and call clear register function
*****************************************/
void INIT()
{
lcd.clear();
lcdtext(0,0,"RESET REGs..");
lcdtext(0,1,"Please Wait...");
clear_REGS();
```

```
delay(d);


lcd.clear();
lcdtext(0,0,"Press RED button");
lcdtext(0,1,"to START");
wait_red();

}



/*****************************************
Read register content
Send READ command
Register address = addr
Return content
*****************************************/
byte READ_REG(byte addr)
{
byte ans;

  Wire.beginTransmission(addr);
  Wire.write(READ);
  Wire.endTransmission();
  delay(200);
  //Receive answer
  Wire.requestFrom(addr, 1);
  if (Wire.available())
    ans = Wire.read();

return ans;
}

/*****************************************
Send command to ALU/SR unit
Address = addr
command to ALU/SR = instruction
Return ALU result
*****************************************/
byte SEND_ALU(byte addr, byte instruction)
{
byte ans;

  Wire.beginTransmission(addr);
```

```
  Wire.write(instruction);
  Wire.endTransmission();
  delay(300);
  //Receive answer (PC content)

  Wire.requestFrom(addr, 1);
  if (Wire.available())
    ans = Wire.read();

return ans;
}


/*****************************************
Load a number to a specific register
Register address = addr
Number = data
*****************************************/
void WRITE_REG(byte addr, byte data)
{

Wire.beginTransmission(addr);
Wire.write(LOAD);
Wire.endTransmission();
delay(200);
Wire.beginTransmission(addr);
Wire.write(data);
Wire.endTransmission();
}
```

CHAPTER 3
System Operation

3.1 HOMS as a system

The previous mentioned components (blocks) have to be reused in order to form a working microprocessor model. The working model consists of General and Special Purpose Registers (GPR-SPR, block-A type), an Arithmetic and Logic Unit (ALU, block-B type), a Control Unit (CU, block-C type) and the memory/output system unit (block-D type). The real model includes eight (8) blocks as registers, one (1) block for Control Unit, one (1) block for ALU and Status Register and one (1) block for the memory/output system. Figure 3.1 shows how the hardware components constitute a working system (microprocessor, memory with user data input and output).

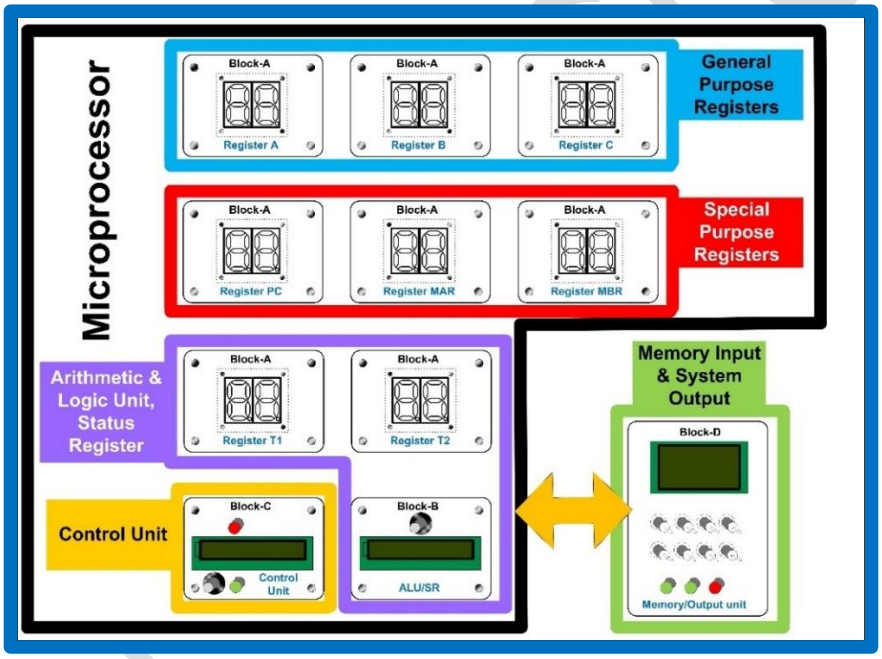


Fig. 3.1 Block organization for a working system

As shown in fig. 3.2, the data transfer between units (e.g. registers, memory) is performed via communication buses. This communication is implemented in the real model by using the I^2C serial protocol. This practical approach is chosen in order to simplify the physical connections between blocks. There are two pairs of common connections between all blocks (SDA-Serial Data, SCL-Serial Clock). Figure 3.2 shows the I^2C connections.

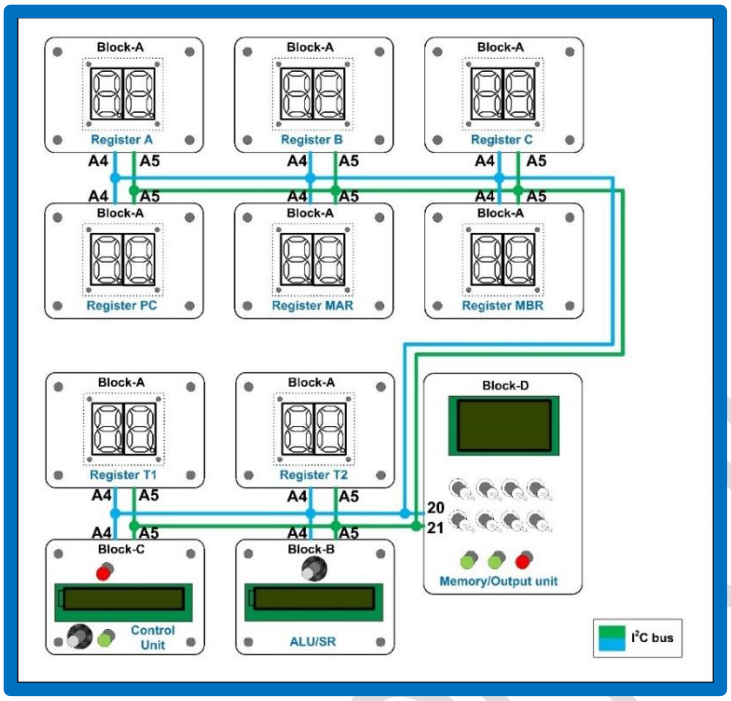


Fig. 3.2 Common connections between blocks

The proposed HOMS tool has been physically installed in a hard aluminum suitcase for supporting mobility and protecting the whole system device from any damage. Figure 3.3 shows the fully working HOMS system.

Fig. 3.3 The final working HOMS tool

3.2 Assembly program execution

As mentioned before, the HOMS tool constitutes a working microprocessor that interacts with the memory unit for executing assembly instructions. All the available HOMS blocks have to communicate to each other via the I^2C bus. The HOMS is based on an 8bit architecture. Thus, registers, memory contents and addresses are all 8bit. The memory unit just holds the instruction data (the first byte for the command and the second byte for the parameter). The control unit ensures that in each execution step, two bytes will be transferred to "microprocessor" starting from the current address that the PC register points to. The value of instruction bytes (command and parameter) makes sense only for the control unit in order to perform the needed actions (instruction execution).
Inside the control unit, an execution loop takes place. Figure 3.4 shows the flow chart for the execution loop. Based on this process, the memory locations are scanned and the control unit executes the corresponding instructions. When a STOP instruction (code value 17) is found, the execution process is terminated. Within the loop, the following tasks are performed:

- *READ_PC*. The starting address of the next instruction to be executed is retrieved from PC register. Every assembly instruction that is stored in memory, has a constant length of two bytes. Thus, the fetching from memory can be implemented in a simple way.
- *PC2MAR.* Inside a microprocessor, the MAR (Memory Access Register) is directly connected to address bus, in order to activate a specific address for reading or writing data. At this step, the content of PC register is copied to MAR.
- *FETCH1.* Based on the MAR content, the first instruction byte is fetched from the memory. This byte is transferred to MBR (Memory Buffer Register) in order to be available to the control unit.
- *UPDATE_MAR.* The content of MAR register is updated (MAR=MAR+1) in order to point to the next address. Thus, the next byte fetching is prepared correctly.
- *FETCH2.* Based on the new MAR content, the second instruction byte is fetched from the memory. This byte is also transferred to MBR (Memory Buffer Register) in order to be available to the control unit.
- *DECODE_EXEC.* Now the command block is completed. The first byte represents the command code and the second byte the parameter. If an assembly instruction has no parameter, this byte is zero but is transferred from the memory based on the above steps.
- *UPDATE_PC.* After instruction execution, the contents of PC register are updated (adding the number 2) for fetching the next instruction from memory.

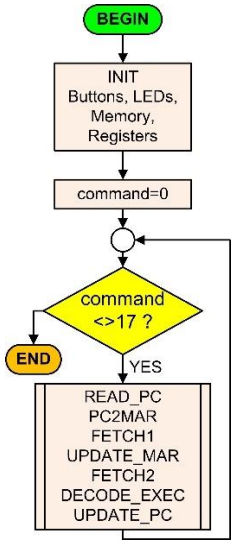


Fig. 3.4 Execution loop

For demonstrating the operation of the HOMS tool, an assembly program is stored in memory and finally executed by the implemented microprocessor model. Table 3.1 shows the demo program (symbolic instruction, byte code and memory contents).

Table 3.1
Demo program

Instruction	Byte code	Address (content) (in decimal)
LOD A,8	(dec) 04 08, (hex) 04 08	00* (04), 01 (08)
INC A	(dec) 10 00, (hex) 0A 00	02* (10), 03 (00)
MOV B,A	(dec) 06 00, (hex) 06 00	04* (06), 05 (00)
ADD A,B	(dec) 14 00, (hex) 0E 00	06* (14), 07 (00)
STOP	(dec) 17 00, (hex) 11 00	08* (17), 09 (00)

* Instruction starting address (PC content)

Table 3.2 shows as an example, how the instruction ADD A,B is executed.

Table 3.2
Execution steps for instruction ADD A,B

Register	STEP										
	1	2	3	4	5	6	7	8	9	10	11
PC	06					DECODE			Addition inside ALU		08
MAR		06		07							
MBR			14		00						
A										[A+B]	
B											
T1							[A]				
T2								[B]			

As shown in table 6, the instruction *ADD A,B* is executed as follows:
STEP 1: The PC shows the starting address of the instruction to be executed (ADD A,B)
STEP 2: The starting address of the instruction is stored in MAR register
STEP 3: The first instruction byte is fetched and is stored in MBR register

STEP 4: The MAR address is increased by one, in order to point to the next address where the second byte of the instruction is stored
STEP 5: The second instruction byte is fetched and is stored in MBR register
STEP 6: The control unit decodes the instruction bytes and starts to execute the instruction
STEP 7: The content of register A is copied in the register T1 which is the first input of the ALU (Arithmetic and Logic Unit)
STEP 8: The content of register B is copied in the register T2 which is the second input of the ALU (Arithmetic and Logic Unit)
STEP 9: The addition T1+T2 is performed inside the ALU
STEP 10: The result is stored in register A
STEP 11: The content of the PC register is updated (increased by two) for pointing to the next instruction in memory

The following figures are indicative of the execution/working process. Figure 3.5 shows the instruction bytes (in decimal) in memory locations.

Fig. 3.5 Program bytes in memory unit

Figure 3.6 is a snapshot of the execution process for the instruction ADD A,B. The PC register point to the starting address 06 where the first byte of the instruction is stored. The last content of MBR is the instruction code (DEC 14, HEX 0E), while the MAR is prepared for fetching the next byte of instruction. The registers A and B contain the same value due to previous instruction execution MOV B,A. After the decoding process, the LCD screen on the control unit displays the full form of the instruction which is under execution (fig. 3.7). After instruction execution, the LCD screen displays the result (fig. 3.8) of the addition T1+T2 (T1=A and T2=B). This result (dec 18, hex 12) is stored back in register A.
Figure 3.9 shows the HOMS status while the STOP instruction (dec 17, hex 11) is under execution for program termination. The register A holds the addition result (dec 9+9=18, hex 9+9=12) and the PC register points to next instruction starting address. Through the register MAR, the address 08 will be accessed for fetching the first byte. After the first fetch, the content of register MBR is 11 (hex) which corresponds to STOP instruction.

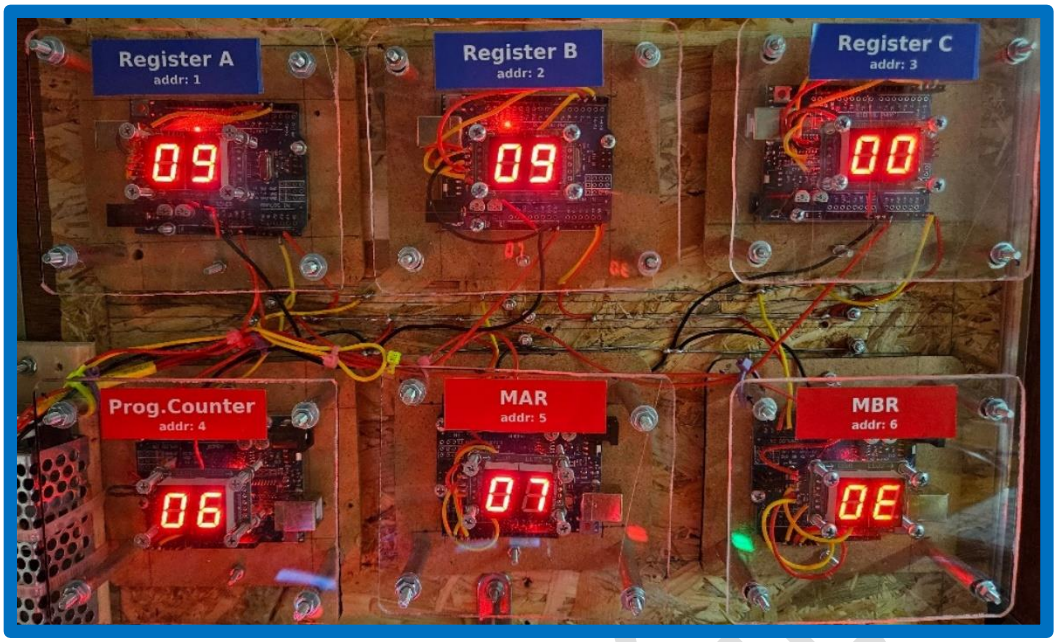


Fig. 3.6 Preparation (MAR) for fetching the second byte of the instruction

Fig. 3.7 Instruction under execution

Fig. 3.8 Addition result in ALU

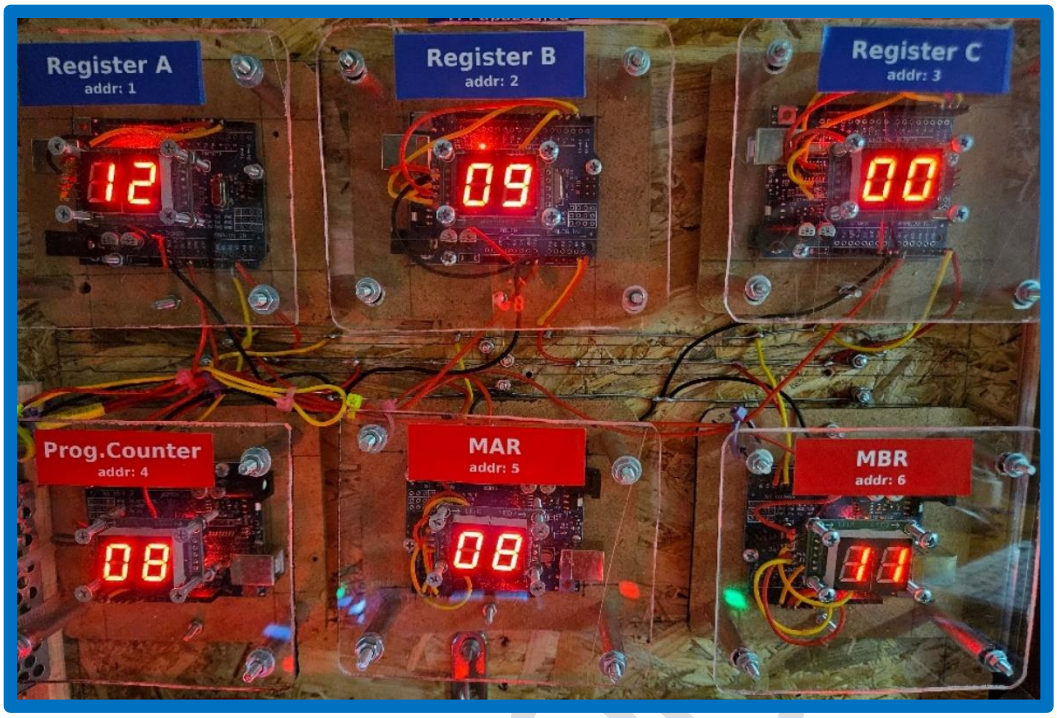


Fig. 3.9 Just before program termination

CHAPTER 4
Build reference

4.1 Design files summary

The HOMS is an open-source educational project and the corresponding material for building, programming, operating and customizing, is freely available to engineering community and other relevant audience. Table 4.1 shows all the available files that are available in order to reproduce, study and use of the proposed HOMS tool.

Table 4.1
Design files

Design file name	File type	Open source license
HOMS-CU.ino Control Unit code	Arduino sketch	CC BY NC SA
HOMS-REG.ino Register Unit code	Arduino sketch	CC BY NC SA
HOMS-ALU-SR.ino Arithmetic & Logic Unit and Status Register code	Arduino sketch	CC BY NC SA
HOMS-MEM-OUT.ino Memory data entry and system output code	Arduino sketch	CC BY NC SA
HOMS-7SD-DEMO.ino Demo code for testing seven segment display module	Arduino sketch	CC BY NC SA
BLOCK-A-WIR.pdf Cable connections	Wiring diagram	CC BY NC SA
BLOCK-B-WIR.pdf Cable connections	Wiring diagram	CC BY NC SA
BLOCK-C-WIR.pdf Cable connections	Wiring diagram	CC BY NC SA
BLOCK-D-WIR.pdf Cable connections	Wiring diagram	CC BY NC SA
SYSTEM-WIR.pdf Wiring diagram of the HOMS tool	Wiring diagram	CC BY NC SA
HOMS-DIM.pdf Component Dimensions	Dimension diagram	CC BY NC SA
COMP-LIST.pdf Component list	All components	CC BY NC SA
SUP_PIC.pdf Supplementary pictures	Various pictures	CC BY NC SA

QUICK-GUIDE.pdf	HOMS basic guide	CC BY NC SA
HOMS-VIDEO.mp4	HOMS working demo	CC BY NC SA
HOMS-VID-BS.mp4	HOMS development video samples	CC BY NC SA

4.2 Bill of materials summary

All the necessary components for building the HOMS tool are very common and can be found in any local or international market. On the other hand, the block dimensions, the lcd screens, the buttons, etc, can be very different as compared to the presented HOMS tool implementation based on designer's choices. Current implementation shows how the concept of the object-oriented approach can be applied. Table 4.2 shows the materials that have been used in HOMS tool as well as the corresponding cost (2023).

Table 4.2
Bill of materials

Designator	Component	Number	Cost per unit (€)	Total cost (€)	Source of materials
Microcontroller (block types A, B, C for control unit, registers, ALU/SR)	Arduino UNO 16MHz (compatible board)	10	8	80	(6)
Microcontroller (block type D for memory and output system)	Arduino MEGA 2560 (compatible board)	1	21	21	(6)
2-digit Display	WHMXE-595-2	8	2.5	20	(6)
LCD Display	1602A	2	3.2	6.4	(6)
LCD/TFT Display	2.4 Inch, LCDTFT	1	11.9	11.9	(6)
Power Supply	5V/5A	1	10	10	(6)
Potentiometer	10KΩ / Linear	2	0.5	1	(6)
Pot knobs	Plastic	2	0.2	0.4	(6)
Illuminated buttons	RED	3	1	3	(6)
Illuminated buttons	GREEN	2	1	2	(6)
Resistor	150Ω, 1/4W, +/- 5%	5	0.01	0.05	(6)
Switch	3 pins	8	0.5	4	(6)
Screws and bolts	For Arduino UNO & MEGA	11	0.01	0.11	(6)

Screws and bolts	For Block	46	0.02	0.92	(6)
Cable (1)	For 5V and SCK (red 22 AWG)	2	0.4	0.8	(6)
Cable (1)	For Ground (black 22 AWG)	1	0.4	0.4	(6)
Cable (1)	SDA connection (yellow 22 AWG)	1	0.4	0.4	(6)
USB cable	Type A to type B, free for every new Arduino board	1	0	0	(6)
MDF wooden base (2)	Blocks A-B-C	10	0.25	2.5	(6)
MDF wooden base (2)	Block D	1	0.35	0.35	(6)
Plexiglass (3)	Blocks A-B-C	10	0.31	3.1	(6)
Plexiglass (3)	Block D	1	0.95	0.95	(6)
OSB wooden base (4)	For all blocks	2	1.05	2.1	(6)
Power cable (5)	Simple/double 2.5m	1	1	1	(6)
Power male plug	Simple	1	1	1	(6)
Power switch	Simple	1	1	1	(6)
Total cost				174.38	

(1) The calculation is based on the cable reel price (7.5m)
(2) The calculation is based on a wooden piece price of 1m x 1m (please choose your desire wood type)
(3) The calculation is based on a plexiglass piece price of 1m x 1m
(4) The calculation is based on a wooden piece price of 1.25m x 0.4m (please choose your desire wood type)
(5) The calculation is based on the cable price per meter (1m)
(6) Electronics and other components can be found in any local or international market. Moreover, many components can be found in rejected materials from other applications and other can be built in the lab.

The suitcase where the HOMS has been installed is optional. Additionally, the designer can choose different cables, buttons, switches, lcd screens, knobs, wood type, etc.

4.3 Build instructions

There is not any complexity regarding the HOMS tool construction. The eight of eleven blocks are identical, and thus, if the designer constructs the first block, then the rest of the procedure is easy due to reusability feature. The wooden base of all blocks has to be drilled in order to place the Arduino above. The plexiglass has to be also drilled for placing the seven segment display. Only few cables are needed for connecting seven segment and LCD displays. On the other hand, the 2.4 inch LCDTFT screen is directly connected on the Arduino MEGA 2560 board. Before the final placement of the constructed blocks,

software and connection tests have to be performed (please use the code *HOMS-7SD-DEMO.ino* for testing the seven segment display module).

Figure 4.1 shows in single steps the construction of a HOMS tool block. Initially, the wooden base is drilled for placing Arduino UNO (fig. 4.1-1, 4.1-2). Next, headers are connected to pins that will be used for controlling the seven segment display module as well as the power lines (fig. 4.1-3). The seven segment module is soldered using five cables (three control cables and two for the power supply) and the final placement is implemented on the plexiglass (fig. 4.1-4, 4.1-5). Finally, the block is completed by placing the four metal spacers (wooden or plastic spacers can be also used) with screws and bolts (fig. 4.1-6). The same construction concept is followed for all blocks in the HOMS tool. The files BLOCK-A-WIR.pdf to BLOCK-D-WIR.pdf show the cable connections for each block and the file *SUP-PICS.pdf* shows a rich picture collection for HOMS construction, components, tools, etc.

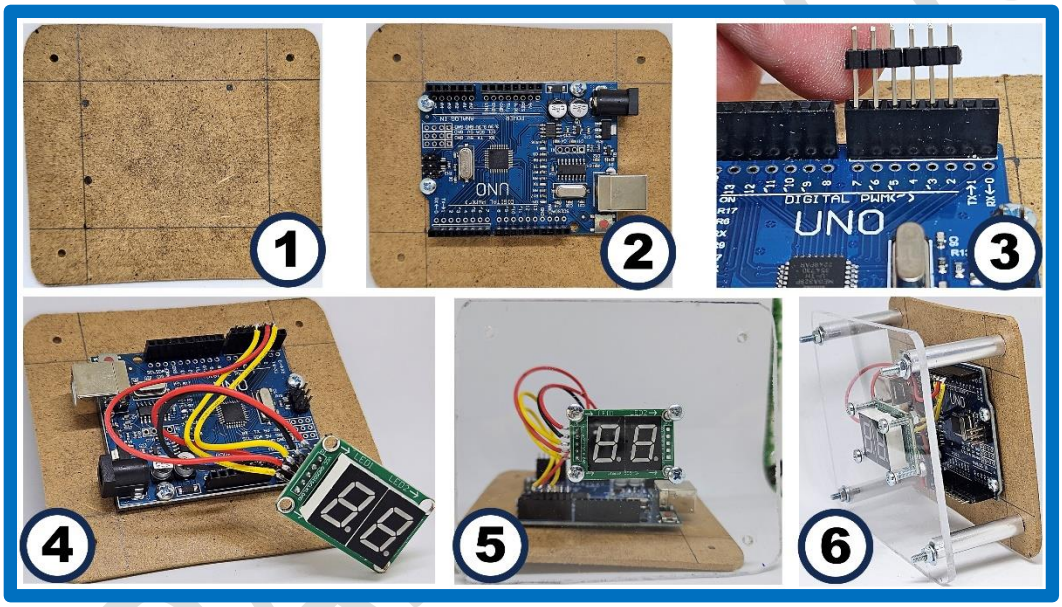


Fig. 4.1 Building a block

For simplifying the common connections (I^2C and GND), three metal wire lines are passed over and under the Arduino boards (fig. 4.2). Thus, the cables from Arduino boards are soldered to common wires (Yellow wire: SDA, Red wire: SCL and Black wire: GND). The metal wires are stretched from point edges using screws that are embedded in the wooden base.

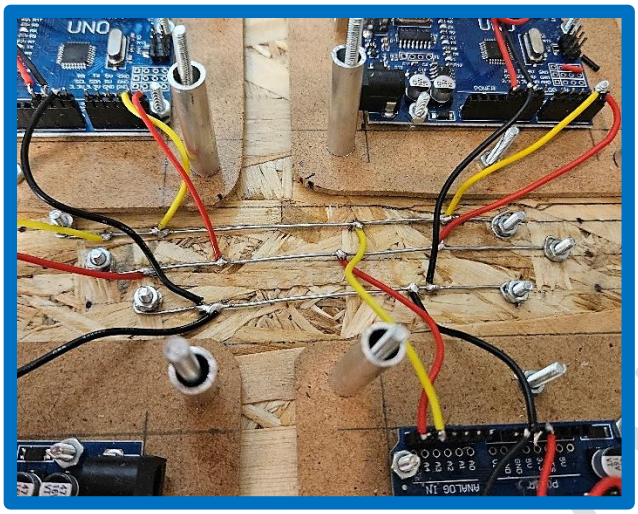


Fig. 4.2 Metal wire lines for common connections

4.4 Operation instructions

After connecting common lines (I^2C and GND), the 5V has to be also connected. The HOMS tool works autonomously and thus no computer is needed. That means that the 5V supply has to be supported by an external power supply (PS). All the Arduino boards (UNO and MEGA) need the same 5V voltage. For supplying Arduino boards with 5V power, only the $\mathbf{V_{in}}$ pin will be used which is a common pin among Arduinos. In the current implementation, an industrial-type PS is used (fig. 4.3). This PS it was already available in the lab of the HOMS tool. A PS with current support up to 1.5A is enough. The PS in the proposed HOMS tool has to be connected directly to 220V power line. For 110V power line, the PS has to support different input. For connecting PS to power line, a single male plug and wire can be used as well as a typical power switch (fig. 4.4).

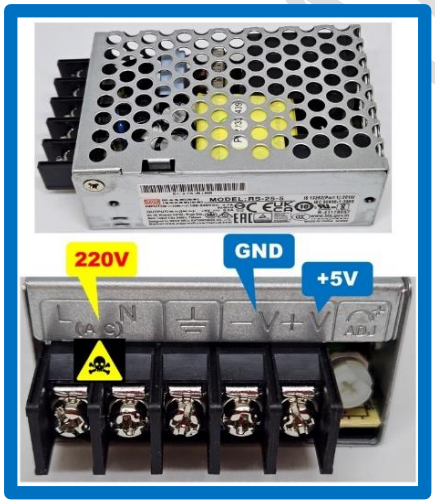


Fig. 4.3 External power supply (PS)

Fig. 4.4 Male plug, cable and switch

Fig. 4.5 Power from a typical USB power supply

If the builder of the HOMS tool is not familiar with the power line connections, a classic USB PS can be used as an alternative solution. As shown in fig. 4.5, a classic USB PS

can be combined with a USB cable. The cable is type-A to type-B, but the end has been cut. The USB cable contains four internal cables with colors. A USB cable “transfers” voltage via the red (5V) and black cable (GND). These internal cables have to be connected to the common HOMS lines for the Arduino pin Vin.

Figure 4.6 shows the connection diagram for the whole HOMS tool regarding PS with or without the industrial PS.

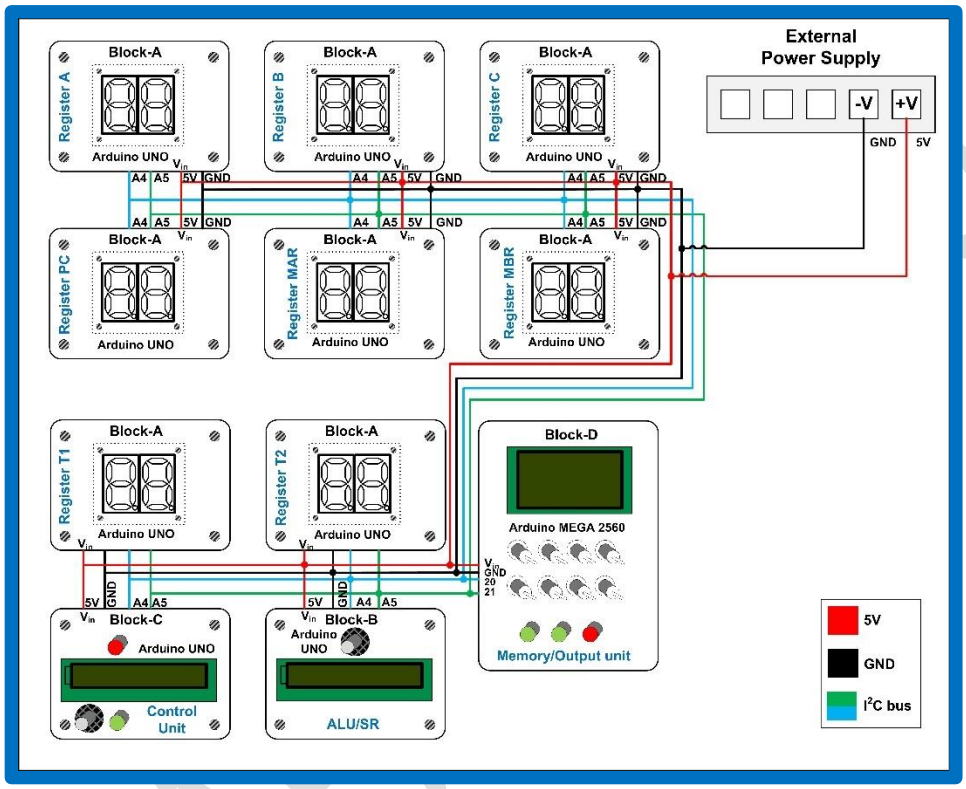


Fig. 4.6 Wiring diagram for I2C and Power Supply connections

About the author [Dr. Panayotis (Panos) Papazoglou]

Associate professor, Department of Digital Arts and Cinema, National and Kapodistrian University of Athens.
He worked also as Lecturer, Assistant Professor and Associate Professor at Technological Educational Institutes of Athens, Lamia and Central Greece, Departments of Electronics, Computer Engineering (Head of Department 2015–2016) and Electrical Engineering respectively. He teaches Computer Architecture and Microprocessor programming for more than 25 years with a total academic experience more than 27 years. Dr P. Papazoglou is the author of 14 scientific-technical books (12 in Greek and 2 in English -Amazon, USA-) and has more than 50 publications in international journals, book chapters and conferences. He is the author of the Greek best seller book "Application Development with Arduino" and the most popular book about microprocessors.

This project did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

P.M.Papazoglou, A Hybrid Simulation Platform for Learning Microprocessors, Computer Applications in Engineering Education, 10.1002/cae.21921, (pp 655-674) WILEY, 2018

Website

https://homs.panospapazoglou.gr/