Trivial Instruction simulation using C

Code:

/\*\*

 \* @file    cpu\_runner.c

 \* @brief   This file contains the algo related to CPU and Memory initialisation and Fetch the Byte-code

 \* @author  Pranjal Chanda (2022HT01011)

 \*/

#include "isa.h"

/\* CPU RAM ATTACHMENT \*/

CPU\_CONTEXT\_t g\_context;

bytecode\_t    g\_code\_mem[CODE\_MEM\_SIZE];

uchar         g\_data\_mem[DATA\_MEM\_SIZE];

/\* Runner \*/

int main(void)

{

    FILE \*f\_ptr;

    uchar byte;

    uchar \*mem\_ptr;

    printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    printf("\* Trivial Instruction Set Simulator using C \*\n");

    printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    printf("[INFO]: Reading Code Memory\n");

    mem\_ptr = (uchar \*)g\_code\_mem;

    /\* Erase Code Mem \*/

    memset(g\_code\_mem, 0xFF, CODE\_MEM\_SIZE);

    memset(g\_data\_mem, 0xFF, DATA\_MEM\_SIZE);

    /\* Read Flash Mem \*/

    f\_ptr = fopen("code.bin","rb");

    if(f\_ptr == NULL)

    {

        perror("File Error!");

        exit(1);

    }

    byte = fgetc(f\_ptr);

    /\* Load The Data to Data Mem \*/

    while(byte != (uchar) EOF)

    {

        \*mem\_ptr = byte;

        byte = fgetc(f\_ptr);

        mem\_ptr++;

    }

    fclose(f\_ptr);

    printf("[INFO]: Reading Data Memory\n");

    mem\_ptr = (uchar \*)g\_data\_mem;

    f\_ptr = fopen("data.bin","rb");

    if (f\_ptr == NULL)

    {

        perror ("File Error!");

        exit (1);

    }

    byte = fgetc(f\_ptr);

    /\* Load The Code-to-code Mem \*/

    while (byte != (uchar) EOF)

    {

        \*Mem\_ptr = byte;

        byte = fgetc(f\_ptr);

        mem\_ptr++;

    }

    close(f\_ptr);

    printf("[INFO]: Reseting PC=0x00\n");

    /\* Reset The Program Counter to code mem offset: 0x00 \*/

    g\_context.PC = g\_code\_mem;

    printf("[INFO]: Staring Pipeline\n");

    while (1)

    {

        if (g\_context.PC >= (g\_code\_mem + CODE\_MEM\_SIZE))

        {

            perror("[ERROR]: PC out of Limits Exception");

            exit(1);

        }

        /\* Fetch and send to Decode \*/

        if(pipeline(\*g\_context.PC, &g\_context))

        {

            /\* Break if Error \*/

            break;

        }

    }

    /\* Save Memory Dump \*/

    f\_ptr = fopen("mem\_op.bin","wb");

    fwrite(g\_data\_mem, sizeof(uchar), DATA\_MEM\_SIZE, f\_ptr);

    fclose(f\_ptr);

    printf("[INFO]: Memory Dump Saved\n");

}

/\*\*

 \* @file    instruction\_parser.c

 \* @brief   This file contains the algo related to the OPCODE and Pipeline to parse it.

 \* @author  Pranjal Chanda (2022HT01011)

 \*/

#include "isa.h"

#include "inttypes.h"

int pipeline(const bytecode\_t bytecode, CPU\_CONTEXT\_t\* context)

{

    /\* Decode \*/

    printf("[DUBUG]: [ PC: %04X R[0-7]:{ ",(uint16\_t) (((uintptr\_t)context->PC - (uintptr\_t)g\_code\_mem) & (uint16\_t)0xFFFF));

    for (size\_t i = 0; i < 8; i++)

    {

        printf("%02Xh ", context->R[i]);

    }

    printf("} B1: %02Xh B2: %02Xh ] -> ", bytecode.byte1, bytecode.byte2);

    switch ( (bytecode.byte1 >> 4) & 0x0F )

    {

        case OP\_MOV\_Rn\_DIR:

            /\* Rn = M[Direct]\*/

            context->R[bytecode.byte1 & 0x0F] = g\_data\_mem[bytecode.byte2];

            printf("MOV R%1u, M[%02X]\n", (bytecode.byte1 & 0x0F), bytecode.byte2);

            break;

        case OP\_MOV\_DIR\_Rn:

            /\* M[Direct] = Rn \*/

            g\_data\_mem[bytecode.byte2] = context->R[bytecode.byte1 & 0x0F];

            printf("MOV M[%02X], R%u\n", bytecode.byte2, (bytecode.byte1 & 0x0F));

            break;

        case OP\_MOV\_MRn\_Rm:

            /\* M[Rn] = Rm \*/

            g\_data\_mem[ context->R[(bytecode.byte2 >> 4) & 0x0F] ] = context->R[bytecode.byte2 & 0x0F];

            printf("MOV M[R%1u], R%u\n", ((bytecode.byte2 >> 4) & 0x0F), (bytecode.byte2 & 0x0F));

            break;

        case OP\_MOV\_Rn\_IMM:

            /\* Rn = Immidiate \*/

            context->R[bytecode.byte1 & 0x0F] = bytecode.byte2;

            printf("MOV R%1u, %02Xh\n", (bytecode.byte1 & 0x0F), (bytecode.byte2));

            break;

        case OP\_ADD\_Rn\_Rm:

            /\* Rn = Rn + Rm \*/

            context->R[(bytecode.byte2 >> 4) & 0x0F] = context->R[(bytecode.byte2 >> 4) & 0x0F] + context->R[bytecode.byte2 & 0x0F];

            printf("ADD R%1u, R%u\n", ((bytecode.byte2 >> 4) & 0x0F), (bytecode.byte2 & 0x0F));

            break;

        case OP\_SUB\_Rn\_Rm:

            /\* Rn = Rn - Rm \*/

            context->R[(bytecode.byte2 >> 4) & 0x0F] = context->R[(bytecode.byte2 >> 4) & 0x0F] - context->R[bytecode.byte2 & 0x0F];

            printf("SUB R%1u, R%u\n", ((bytecode.byte2 >> 4) & 0x0F), (bytecode.byte2 & 0x0F));

            break;

        case OP\_JZ\_Rn\_REL:

            /\* Set PC (Jump) if Rn is Zero \*/

            context->PC += context->R[bytecode.byte1 & 0x0F] == 0 ? (char)bytecode.byte2 : 1;

            printf("JZ  R%1u, %02Xh\n", (bytecode.byte1 & 0x0F), (char)bytecode.byte2 & 0xFF);

            return 0;

        case OP\_JNZ\_Rn\_REL:

            /\* Set PC (Jump) if Rn is not Zero \*/

            context->PC += context->R[bytecode.byte1 & 0x0F] != 0 ? (char)bytecode.byte2 : 1;

            printf("JNZ R%1u, %02Xh\n", (bytecode.byte1 & 0x0F), (char)bytecode.byte2 & 0xFF);

            return 0;

        default:

            printf("Case EOF\n");

            return 1;

    }

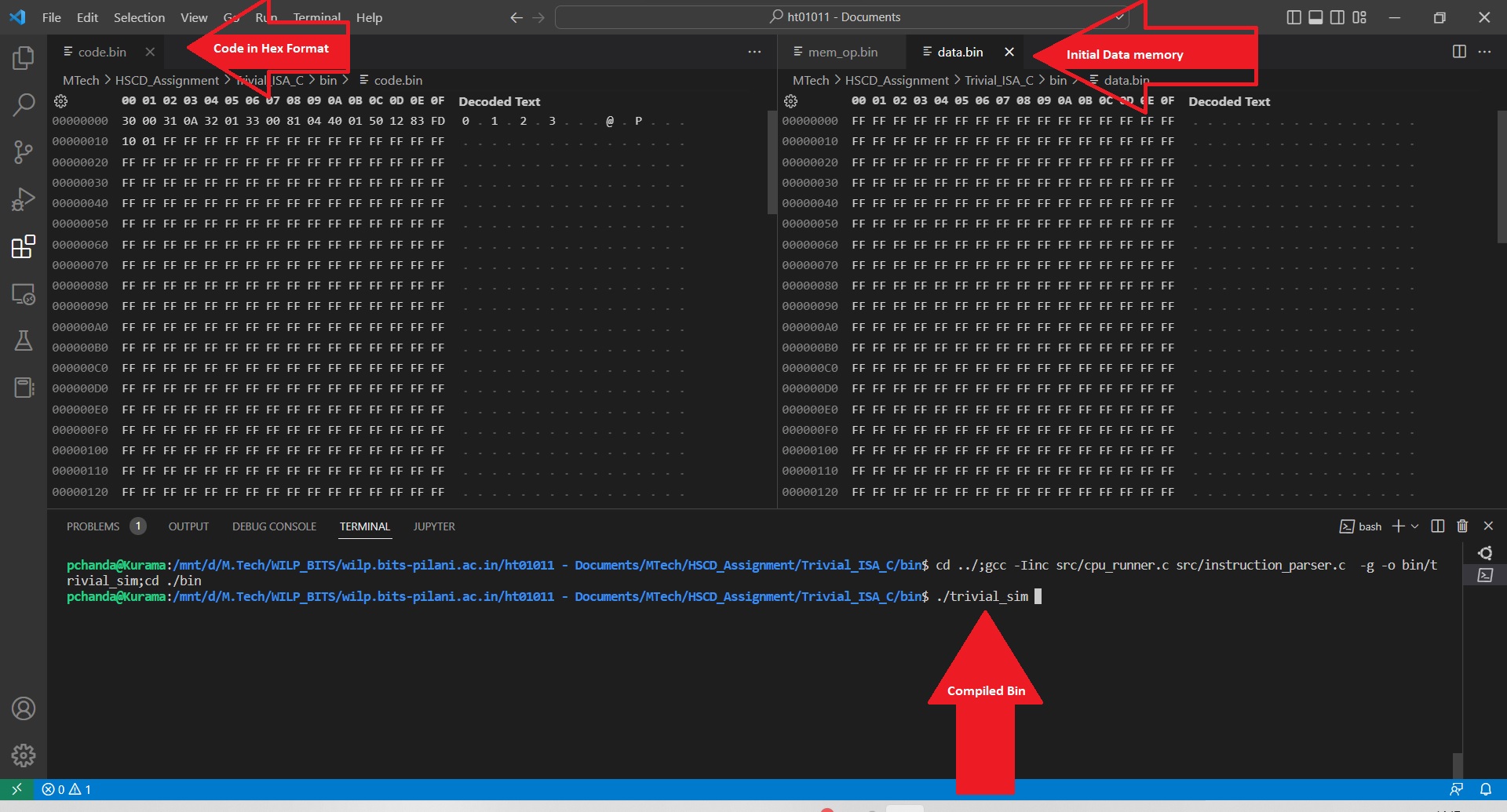
    context->PC++;

    return 0;

}

Compilation:

Files:

1. code.bin consist of application code that is to be simulated in Hex format. Hence, it is the self-compiled binary of Addition of 10 numbers.
2. data.bin consist of data memory contents i.e., constant values
3. mem\_op.bin is the output file that shows the current memory values.

Simulation Output:

**pchanda@pchanda:/mnt/HSCD\_Assignment/Trivial\_ISA\_C/bin$**./trivial\_sim

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Trivial Instruction Set Simulator using C \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[INFO]: Reading Code Memory

[INFO]: Reading Data Memory

[INFO]: Resetting PC=0x00

[INFO]: Staring Pipeline

[DUBUG]: [ PC: 0000 R[0-7]:{ 00h 00h 00h 00h 00h 00h 00h 00h } B1: 30h B2: 00h ] -> MOV R0, 00h

[DUBUG]: [ PC: 0002 R[0-7]:{ **00h** 00h 00h 00h 00h 00h 00h 00h } B1: 31h B2: 0Ah ] -> MOV R1, 0Ah

[DUBUG]: [ PC: 0004 R[0-7]:{ 00h **0Ah** 00h 00h 00h 00h 00h 00h } B1: 32h B2: 01h ] -> MOV R2, 01h

[DUBUG]: [ PC: 0006 R[0-7]:{ 00h 0Ah **01h** 00h 00h 00h 00h 00h } B1: 33h B2: 00h ] -> MOV R3, 00h

[DUBUG]: [ PC: **0008** R[0-7]:{ 00h 0Ah 01h **00h** 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 00h 0Ah 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **0Ah** 0Ah 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 0Ah **09h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 0Ah 09h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 0Ah 09h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **13h** 09h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 13h **08h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 13h 08h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 13h 08h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **1Bh** 08h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 1Bh **07h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 1Bh 07h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 1Bh 07h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **22h** 07h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 22h **06h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 22h 06h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 22h 06h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **28h** 06h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 28h **05h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 28h 05h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 28h 05h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **2Dh** 05h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 2Dh **04h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 2Dh 04h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 2Dh 04h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **31h** 04h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 31h **03h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 31h 03h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 31h 03h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **34h** 03h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 34h **02h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 34h 02h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 34h 02h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **36h** 02h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

[DUBUG]: [ PC: 000E R[0-7]:{ 36h **01h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 36h 01h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **000A** R[0-7]:{ 36h 01h 01h 00h 00h 00h 00h 00h } B1: 40h B2: 01h ] -> ADD R0, R1

[DUBUG]: [ PC: 000C R[0-7]:{ **37h** 01h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2

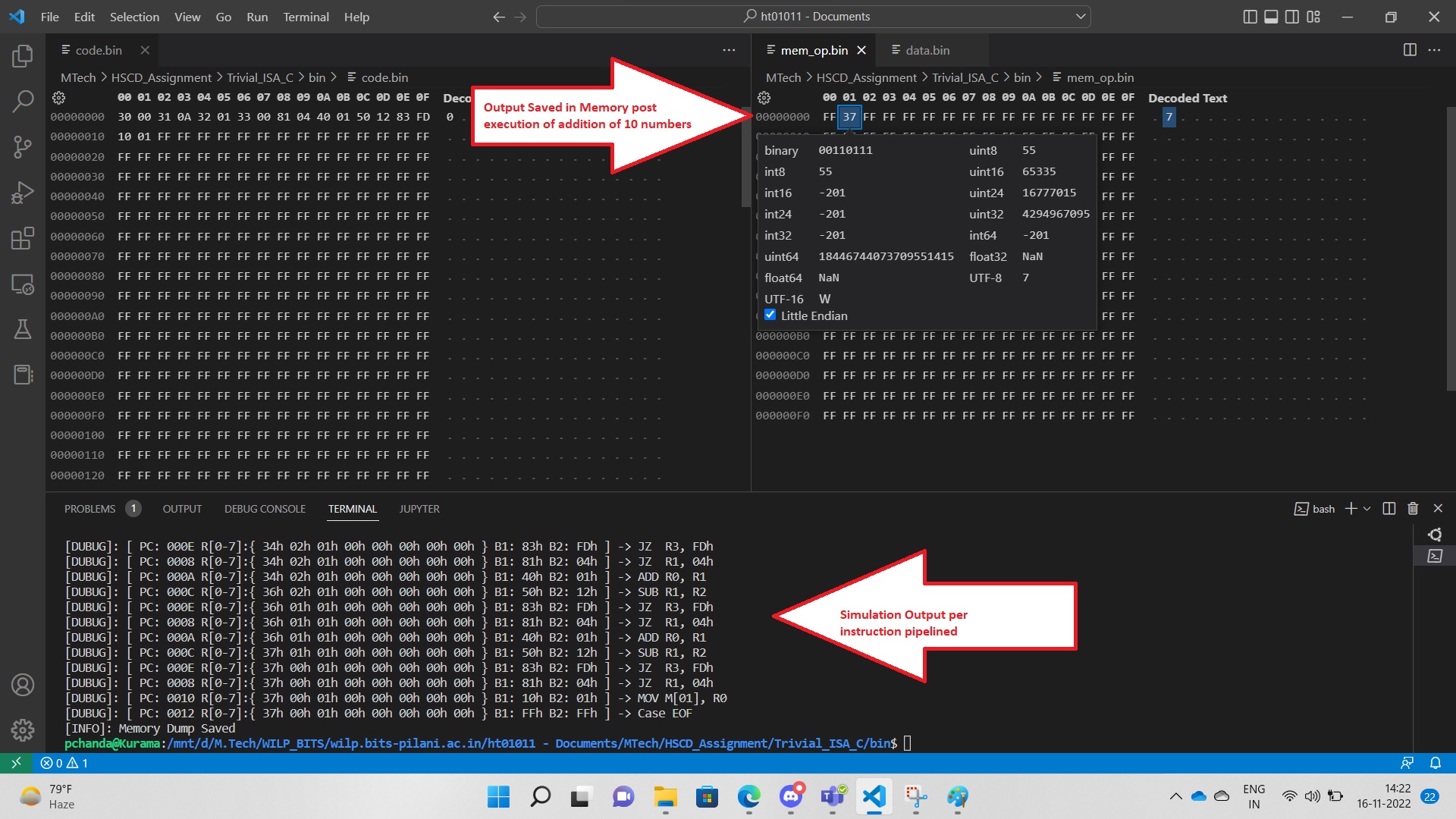
[DUBUG]: [ PC: 000E R[0-7]:{ 37h **00h** 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ R3, FDh

[DUBUG]: [ PC: **0008** R[0-7]:{ 37h 00h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ R1, 04h

[DUBUG]: [ PC: **0010** R[0-7]:{ **37h** 00h 01h 00h 00h 00h 00h 00h } B1: 10h B2: 01h ] -> MOV **M[01]**, R0

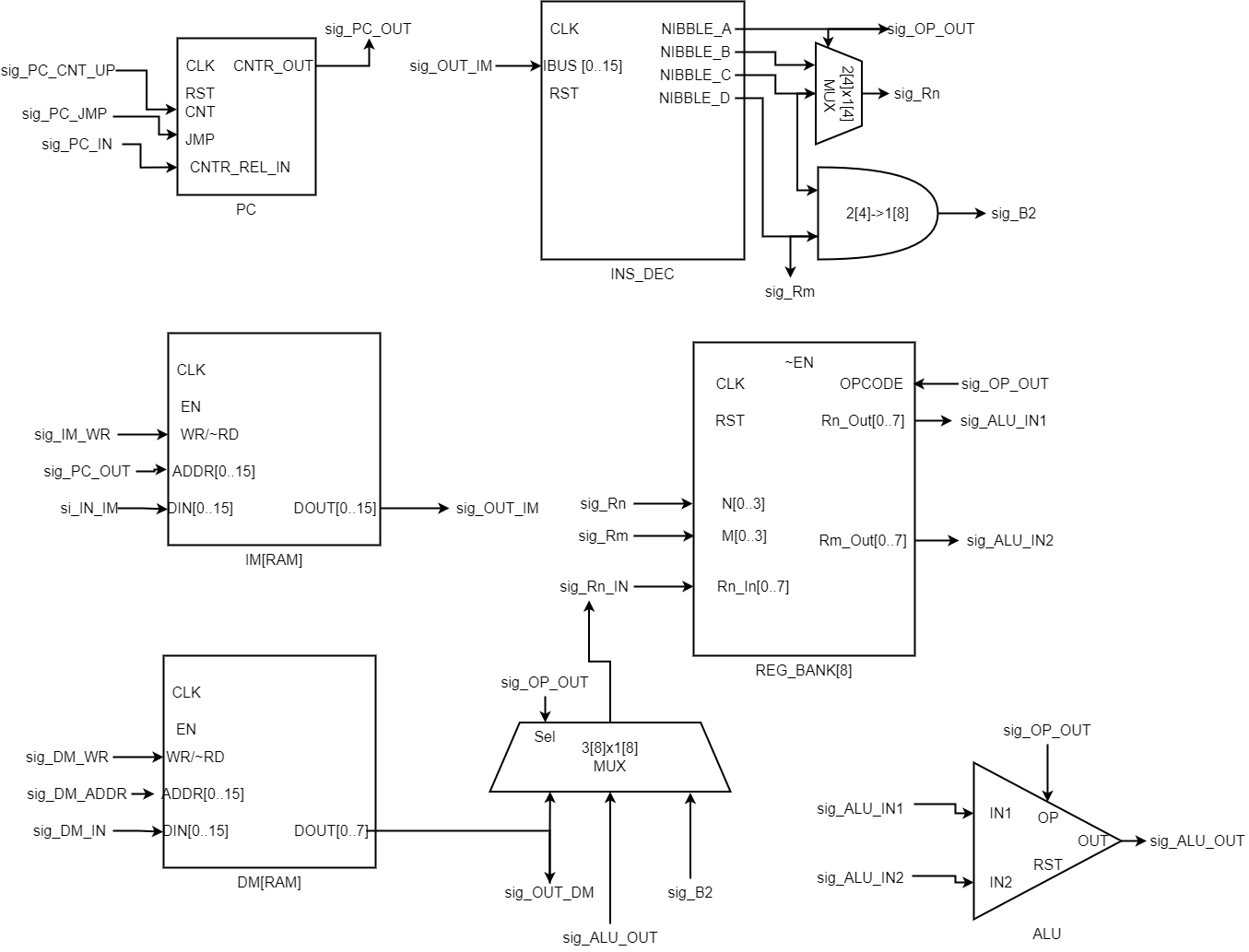
[DUBUG]: [ PC: **0012** R[0-7]:{ 37h 00h 01h 00h 00h 00h 00h 00h } **B1: FFh B2: FFh** ] -> **Case EOF**

[INFO]: Memory Dump Saved



Trivial Instruction simulation using SystemC

CPU Architecture Diagram:



The architecture consists of the following components:

1. Program Counter x1
2. Instruction Decoder x1
3. ALU x1
4. Instruction Memory[RAM] x1
5. Data Memory[RAM] x1
6. Register Bank [8] x1
7. MUX x2
8. Adder x1

Hardware Flow:

1. The Instruction and Data memory is initialised with Code and Data respectively.
2. The Program counter set to Zero.
3. Fetch Instruction from IM as per the address provided by PC and send it to Instruction Decoder.
4. The ID divides the instructions into 4 bit nibbles that is further sent to the following according to the OPCODE provided by IM.
   1. ALU
   2. REG\_BANK
   3. Data Memory
   4. Program Counter
5. Loop continues till the IM OPCODE is invalid