Trivial Instruction simulation using C

Code:

```c
/**
 * @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;
```

```c
/* 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]*/
```

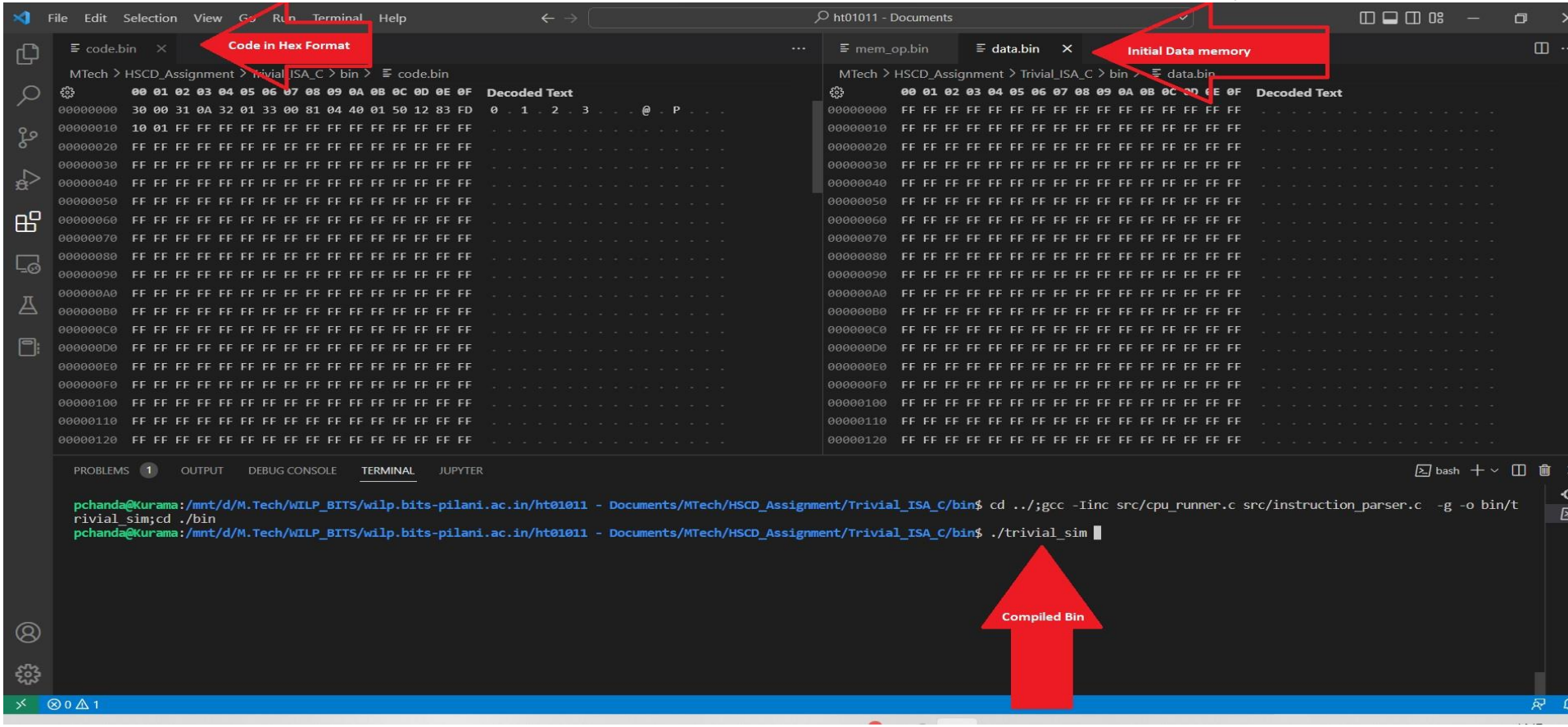
```
        context->R[bytecode.byte1 & 0x0F] = g_data_mem[bytecode.byte2];
        printf("MOV R%1u, M[%02X]\n", (bytecode.byte1 & 0x0F), bytecode.byte2);
        break;
    case OP_MOV_DIR_Rn:
        /* M[Direct] = Rn */
        g_data_mem[bytecode.byte2] = context->R[bytecode.byte1 & 0x0F];
        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.

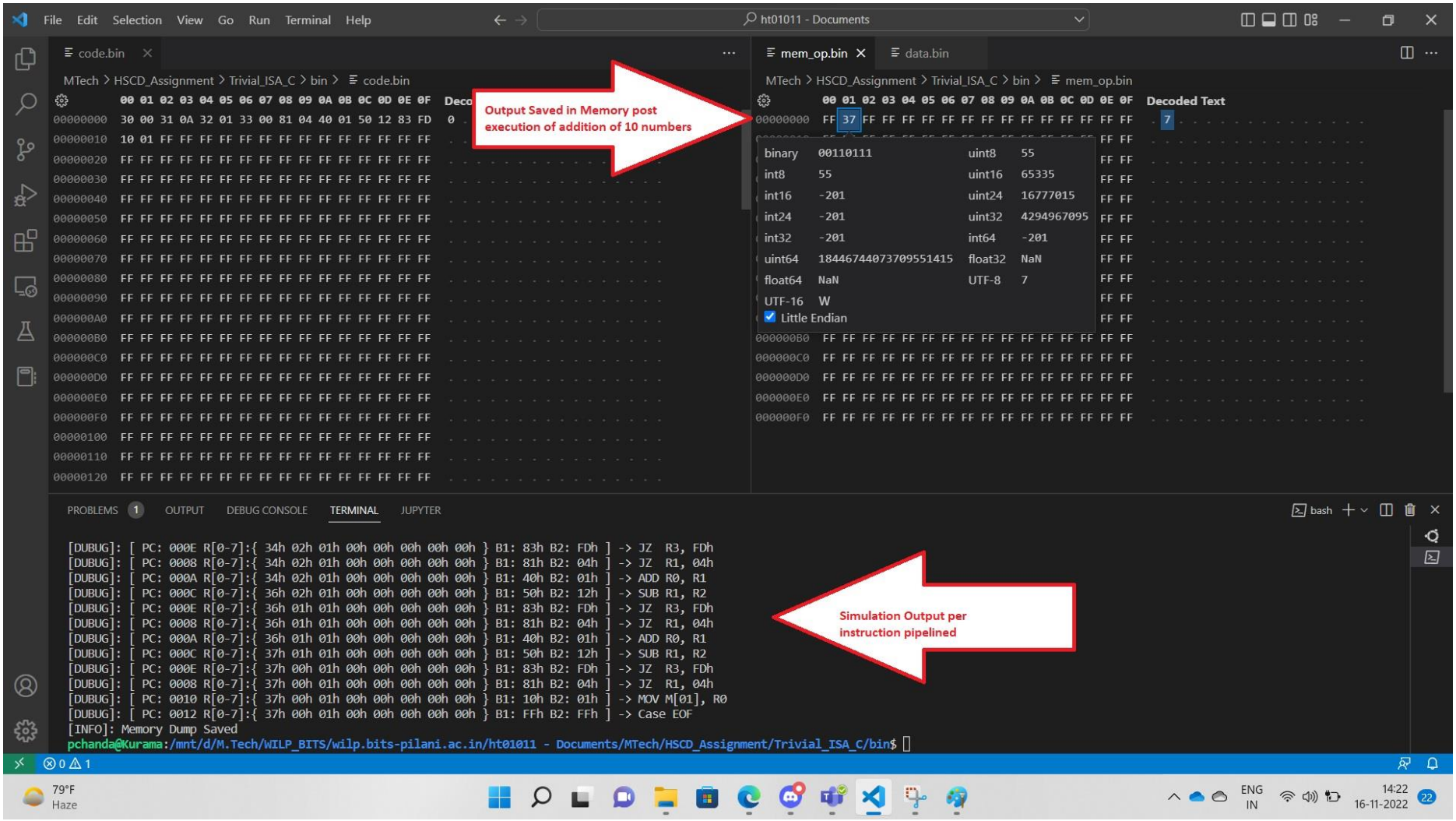


Simulation Output:

```
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
```

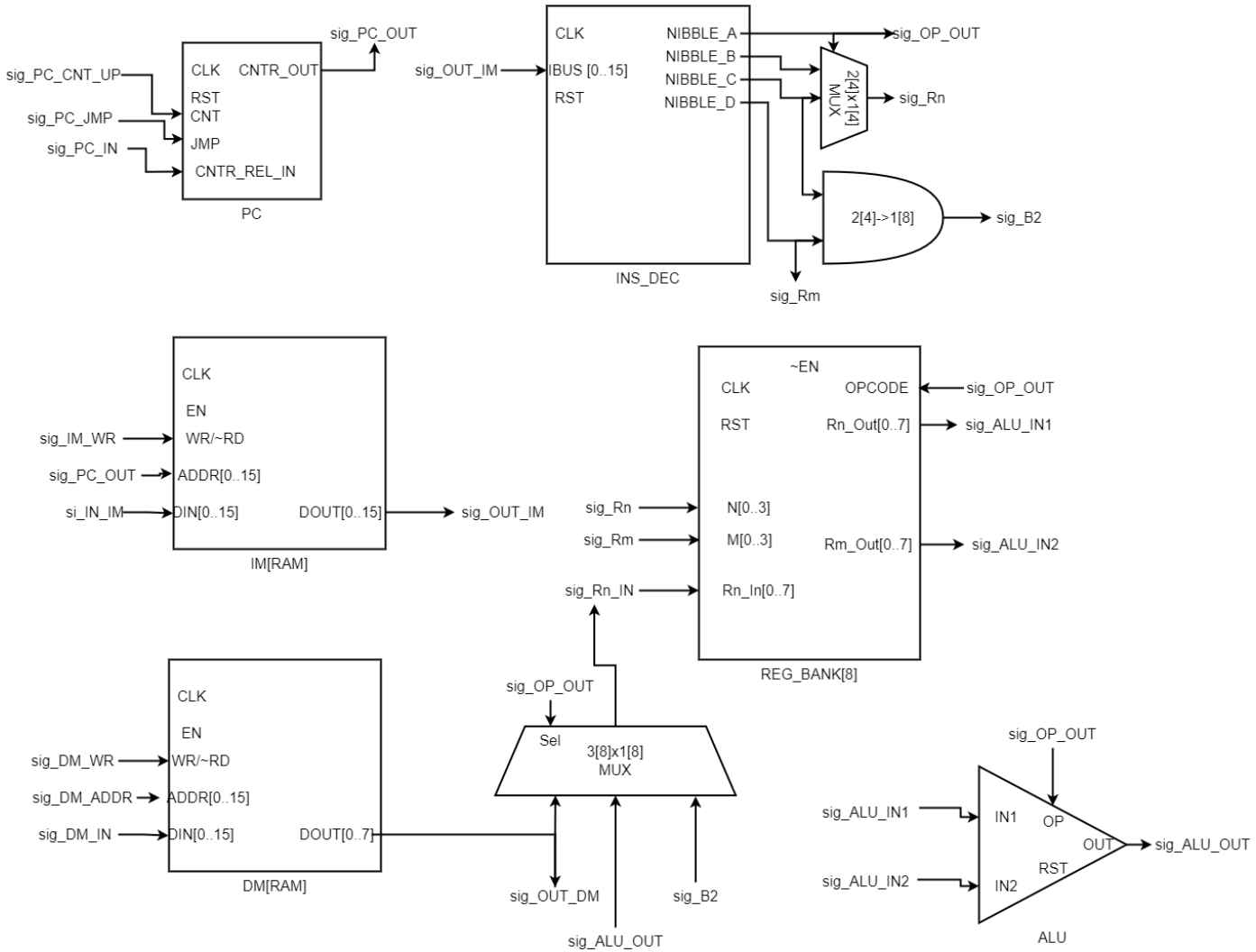
```
[DUBUG]: [ PC: 000C R[0-7]:{ 13h 09h 01h 00h 00h 00h 00h 00h } B1: 50h B2: 12h ] -> SUB R1, R2
[DUBUG]: [ PC: 000E R[0-7]:{ 13h 08h 01h 00h 00h 00h 00h 00h } B1: 83h B2: FDh ] -> JZ  R3, FDh
[DUBUG]: [ PC: 0008 R[0-7]:{ 13h 08h 01h 00h 00h 00h 00h 00h } B1: 81h B2: 04h ] -> JZ  R1, 04h
[DUBUG]: [ 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
```


File Edit Selection View Go Run Terminal Help
ht01011 - Documents
code.bin
mem_op.bin
data.bin
MTech > HSCD_Assignment > Trivial_ISA_C > bin > code.bin
MTech > HSCD_Assignment > Trivial_ISA_C > bin > mem_op.bin
Output Saved in Memory post execution of addition of 10 numbers
Decoded Text
binary 00110111 uint8 55
int8 55 uint16 65335
int16 -201 uint24 16777015
int24 -201 uint32 4294967095
int32 -201 int64 -201
uint64 18446744073709551415 float32 NaN
float64 NaN UTF-8 7
UTF-16 W
Little Endian
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
bash
[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
pchanda@Kurama:/mnt/d/M.Tech/WILP_BITS/wilp.bits-pilani.ac.in/ht01011 - Documents/MTech/HSCD_Assignment/Trivial_ISA_C/bin$
Simulation Output per instruction pipelined
79°F Haze
ENG IN
14:22 16-11-2022

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.
    a. ALU
    b. REG_BANK
    c. Data Memory
    d. Program Counter
5. Loop continues till the IM OPCODE is invalid