# 16 – Bit Microprocessor using FPGA

- DRS2 : Samuel Priyadarshan S

## Abstract:

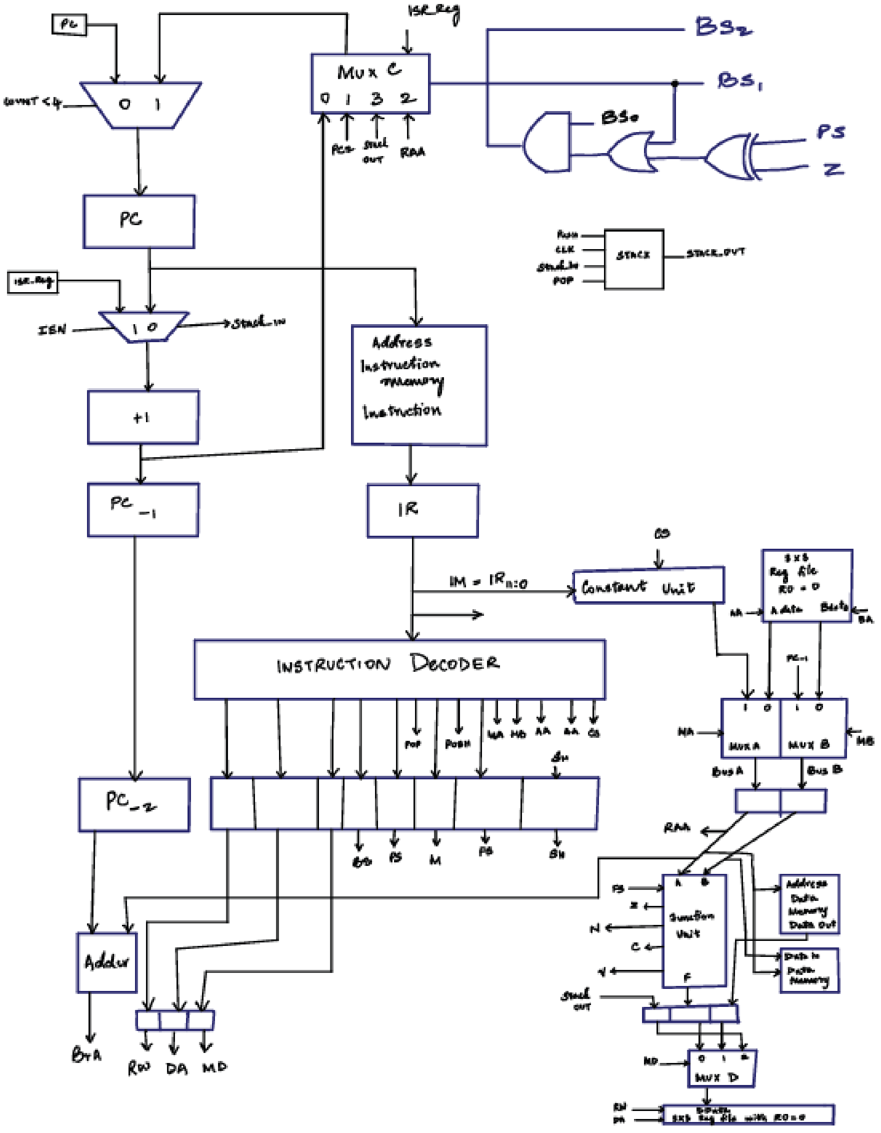
Embedded microprocessors have been widely used as a tool for technological innovations and cost reduction. Its speed and programmability are the main characteristics determining its performance. Therefore, for a design to be competitive, its processor must fit the following characteristics: relatively inexpensive, flexible, adaptable, fast, and reconfigurable. A solution to this is the use of Field-programmable gate arrays (FPGA) as a design tool. The objective of this project is to design a 16-bit customizable microprocessor with components like Program Memory, Data Memory, Register File, ALU, Instruction Decoder, Data Hazard Stall, MUXs, Branch Detection, and Branch Combinational logic.

An ASIC is built for a specific application and the design is hardened for one specific application. With FPGA, the configurable logic blocks and programmable interconnects have been used for programming and reprogramming the FPGAs to perform functions. Soft IPs have been employed for the reprogramming of the FPGAs for a specific application.

## Table of Contents:

The following report contains the following sections in the following order,

# Architecture Block Diagram:



# Features of the microprocessor:

- A 16-bit data bus and 12-bit address bus that enables direct access to up to 4096 16-bit memory locations
- 12-bit program counter (PC) and 12-bit stack pointer (SP) are not accessible to users.
- Single bit carry (C), zero (Z), and interrupt enable (IEN)
- Memory-mapped input/output for communication with the input and output devices

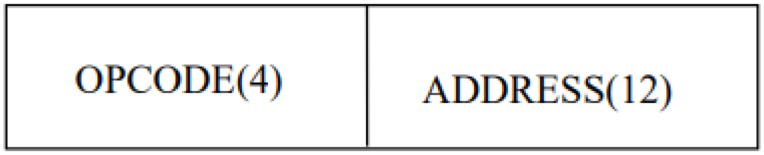
- Load/Store microprocessor architecture with a simple instruction cycle consisting of four machine cycles per instruction cycle
- All data transformations are performed in working registers
- Support for direct and the most basic stack addressing mode, as well as implicit addressing mode
- Definable custom instructions and functional blocks which execute custom instructions can be added and implemented using an FPGA

# Instruction Set:

- All instructions are 16-bits long and require one memory word
- The basic instruction set is as follows

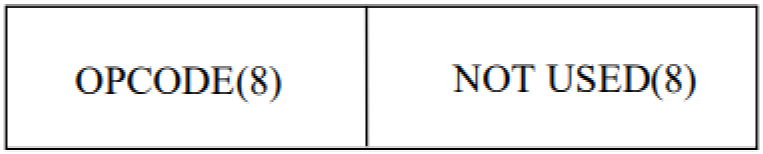
| **Mnemonic** | **Function** |
|---|---|
| LDA | A ←M[address] |
| LDB | B ←M[address] |
| STA | M[address] ← A |
| STB | M[Address] ← B |
| JMP | PC ← Address |
| JSR | Stack ← PC, PC ← address,<br>SP ← SP-1 |
| ADD | A ← A+B |
| AND | A ← A AND B |
| CLA | A ← 0 |
| PUSHA | Stack ← A, SP ← SP-1 |
| POPA | SP ← SP+1, A ← stack |
| CLB | B ← 0 |
| CMB | B ← B’ |
| INCB | B ← B+1 |
| DECB | B ← B-1 |
| CLflag | Flag ← 0 (flag can be carry, zero) |
| ION | IEN ← 1, enable interrupts |
| IOF | IEN ← 0, disable interrupts |
| SZ | If Z=1, PC ← PC+1; skip if zero is set |
| SC | If C=1, PC ← PC+1; skip if carry set |
| RET | SP ← SP+1, PC ← stack |

- All Memory reference instructions use either direct or stack addressing modes and have the format as shown below:



| | |
|---|---|
| OPCODE(4) | ADDRESS(12) |

- Some examples of these instructions are LDA, LDB, PUSHA, POPA etc.
- The remaining core instructions have the following instruction formats:



| | |
|---|---|
| OPCODE(8) | NOT USED(8) |

- Some examples of these instructions are ADD, INCB, DECB etc.
- Instruction execution is divided into 3 major steps along 4 machine clock cycles.
  - ➔ Instruction Fetch- The first cycle, T0, is used to transfer the address of the next instruction from the program counter to the address register. The second cycle T1 is used to actually read the instruction from the memory location into instruction register, IR. At the same time program counter is incremented by one to the value that usually represents the next instruction address.
  - ➔ Instruction Decode - Instruction decode is the recognition of the operation that has to be carried out and the preparation of effective memory address. This is done in the third machine cycle T2 of the instruction cycle.
  - ➔ Instruction Execution - Instruction execution is when the actual operation specified by the operation code is carried out. This is done in the fourth machine cycle T3 of instruction cycle.
- Our reference architecture has been taken from the book by M. Morris Mano.
- The various blocks present in our microprocessor design are as follows.

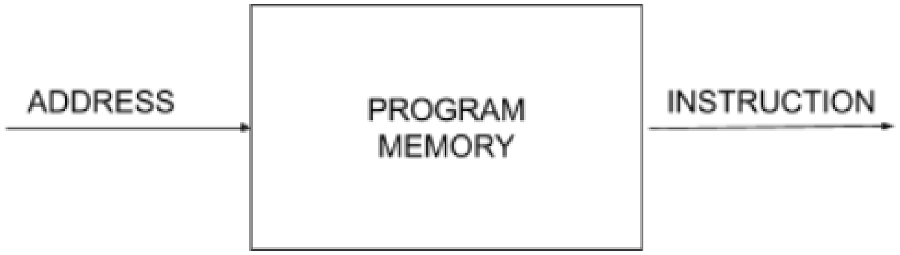
**HARDWARE DESCRIPTION:**

**PROGRAM MEMORY:**

- The Program Memory is the module that contains all the instructions that the MCU can perform.
- The value from the program counter goes into the Program Memory. The program counter gives an 8-bit address value which can access 256 memory locations in the Program Memory.

- Each memory unit consists of a 16-bit value that contains the instructions to be executed.

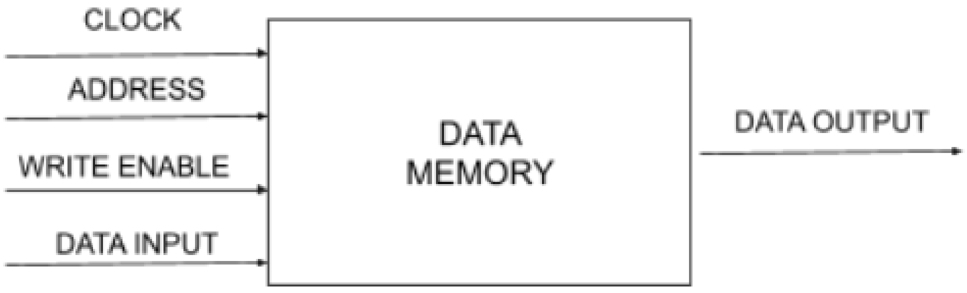
**MODULE DIAGRAM FOR PROGRAM MEMORY:**



**DATA MEMORY:**

The data memory is the memory unit into which the values can be stored. The data memory consists of a write enable, clk, data input and address as inputs, and a data output as output. This module works with the clock. Whenever the write enable is set to 1, the module writes the value in data input into the address provided. Else, it just reads the value in the address provided and gives it as the output.

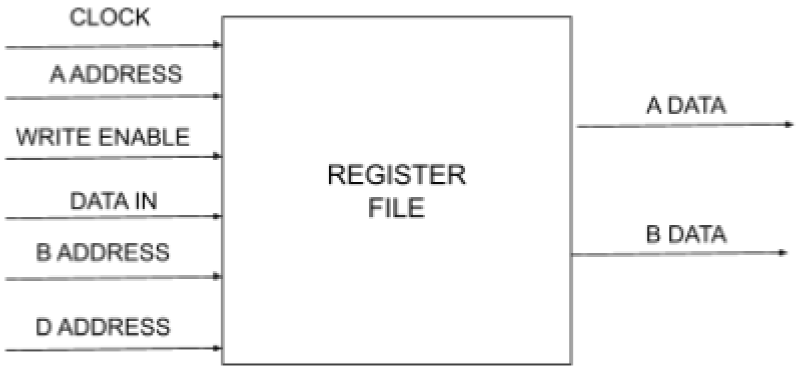
**MODULE DIAGRAM FOR DATA MEMORY:**



**REGISTER FILE:**

The register file is a module that has been designed to contain 8 registers, each of which can store an 8-bit value in it. Of the 8 registers, register 0 always has the value 0 stored in it. The operation is similar to that of data memory. The module stores a value into the register when the read-write enable is set to 1. Else it just reads the value of registers based on the address provided and gives them out as the output.

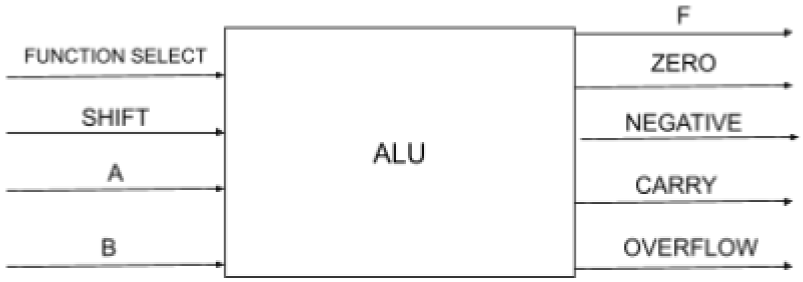
## MODULE DIAGRAM FOR REGISTER FILE:



## ALU:

The ALU is the module which handles all the arithmetic and logical operations. It consists of a Function select, shift, A and B as the inputs and F, zero, negative, carry and overflow as the outputs. The outputs contain 4 flags which are used in different operations such as branch at zero or branch at non-zero. The designed ALU performs operations such as addition, subtraction, Exclusive-OR, OR, NOT, AND, set if less than and logical shift left.

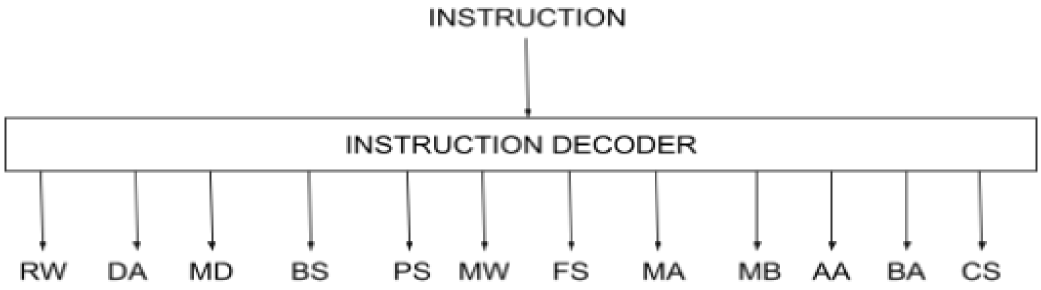
## MODULE DIAGRAM FOR ALU:



## INSTRUCTION DECODER:

The instruction decoder is the main brain of the MCU. The Instruction Decoder plays a vital role in deciding what operations should take place for a particular instruction. It makes sure if the read-write of the register file should be set, operation of various MUXs etc. The input for this module is a 17-bit instruction and has 12 outputs namely read-write of the register file, data address, MUX D sel, branch select, zero toggles, memory write, function select, MUX A sel, MUX B sel, Register A address, Register B address, and constant select.

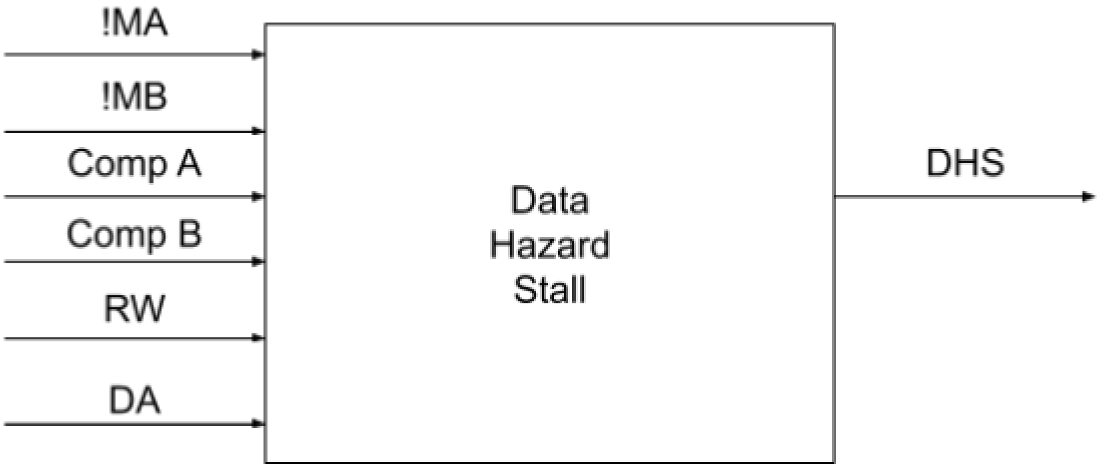
## MODULE DIAGRAM FOR INSTRUCTION DECODER:



## DATA HAZARD STALL:

The Data Hazard Stall is a concept which is implemented to prevent an invalid data getting written in the write back stage. When an operation is using a register, it is likely that the next instruction might use the same register as its operand aswell. But as various stages of the pipeline might take different clock cycles to complete its writeback stage to occur, the fetch cycle may take a wrong data as its operands. Hence in situations like this a Data Hazard Stall can be crutial. When the DHS is set, it makes sure that the value doesn't get written into the register file or data memory

## MODULE DIAGRAM FOR DATA HAZARD STALL:



**BRANCH DETECT:**

Branch Detection is a concept that works similarly to a DHS. When branch select is nonzero, then the compliment of the BS value which is zero is given as the input for RW, MW, and BS to avoid any values written into the register or data memory.

**MP_MAIN:**

Once all these modules were developed, these modules have to be stitched together to work synchronously with the clock of the MCU. There are 4 MUXs namely MUX A, MUX B, MUX C and MUX D which must be designed and various registers and wires that must be initialized to connect various components of the MCU together. There are 3 program counter registers that are working 1 clock cycle behind each other. The IR register gets the 17 bit value from the program memory. There are 2 registers that save the value of register A and B. All these are connections are connected to work synchronously together.

**Implementation:**

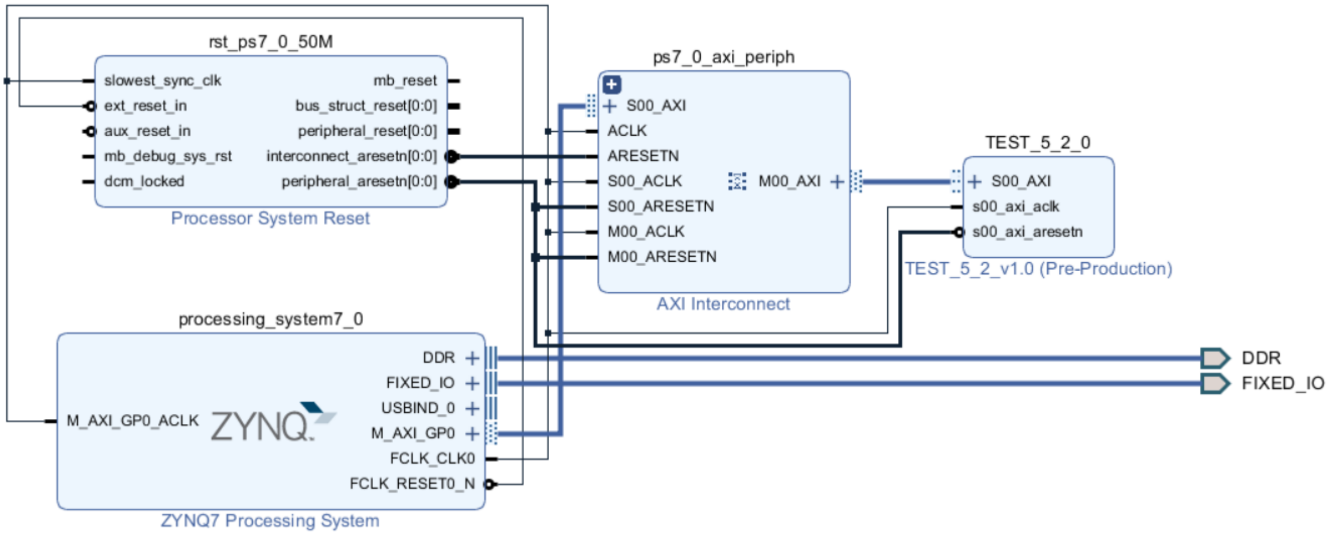
The design implemented is a 16-bit RISC Harvard Architecture. This design is adopted from the architectural design mentioned in the book "*M. Morris R. Mano, Charles R. Kime, Tom Martin - Logic and computer design fundamentals-Prentice Hall (2015)*". However, this architecture cannot work for the given set of operations and hence changes had to be made to ensure proper working of the main module. The revised architecture has been show in this report. We tried implementing data-hazard stall to ensure that a read doesn't happen on the same register before it could write back. But, the design dint work. The pipelined architecture was also tried to parallelize, but the desired output was not obtained just because of the failure of implementation of data hazard stalling. Hence, this design only executes one instruction at a time.

# CUSTOM IP GENERATION

In vivado, once the design has been comfortably tested, this design has to be packed into a custom IP so that it can be attached to the ZYNQ Processor and tested in SDK. For this the main module has to be imported and instantiated into the S00_AXI custom IP verilog file. Upon using the 2 out of the 4 registers provided in the template, then it could be packed and then made into an IP

# BLOCK DIAGRAM OF THE CIRCUIT

This custom IP is then included into the repository of vivado and then the usual procedure of automating connections is followed. Then the bitstream file is generated and is exported as a hardware .hdf file.



# ILA / HW & SW RESULTS OF VARIOUS TEST CASES

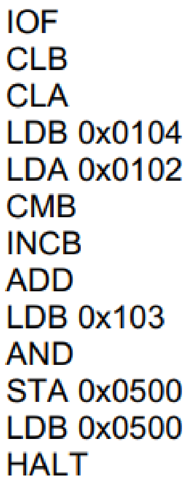
This exported HW file is then tested with a C code. The “xparameters.h” provides the base address of the S_AXI peripheral which is then access ans printed on the serial monitor which is out register A and B

## ARITHMETIC COMPUTATION IMPLEMENTATION:

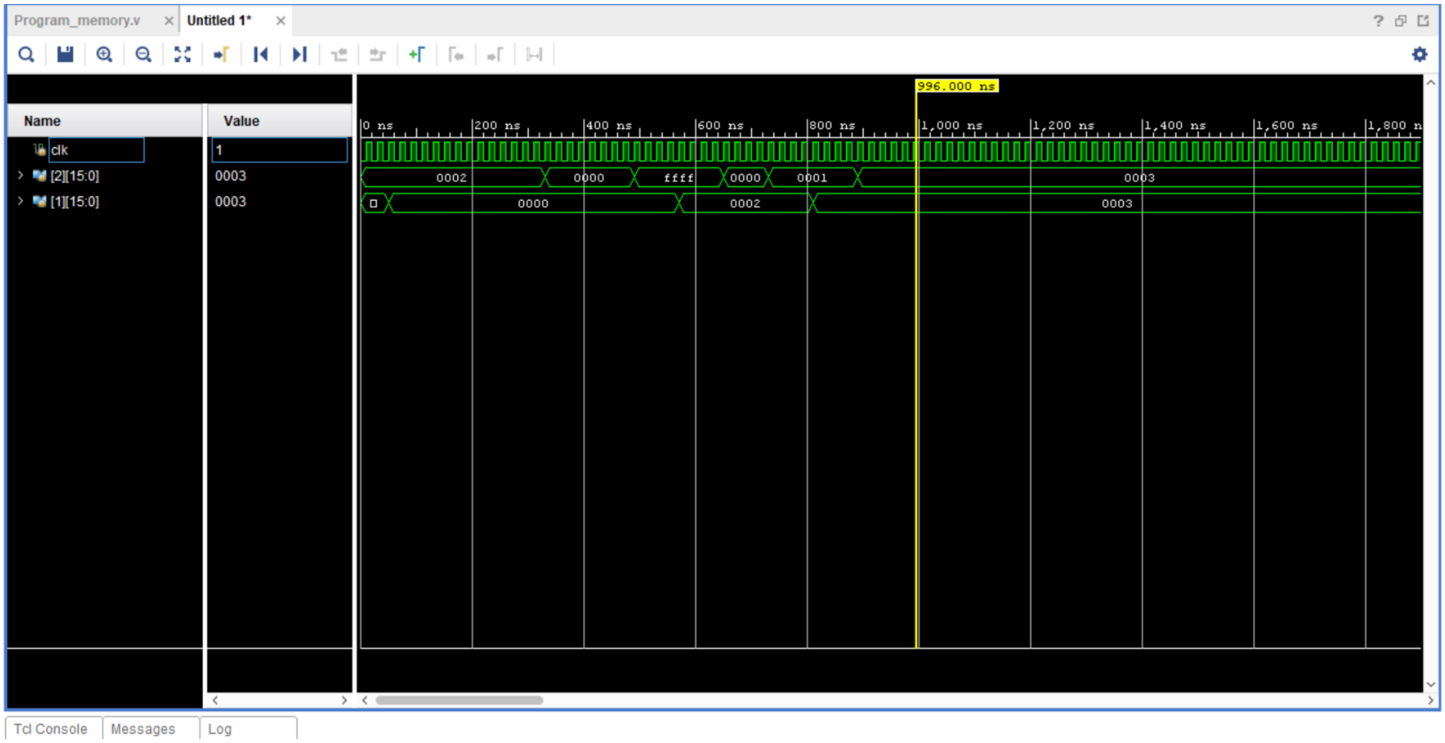
- The arithmetic computation implemented in the design include
    - ADD
    - Negate
    - Left shift
    - Increment A
    - Increment B
    - Clear A
    - Clear B, etc.
- The precision used for the implementation of the 16-bit is integer represented in hexadecimal.
- The input data and the output data are represented in hexadecimal.

### SIMULATION:

TEST_1:



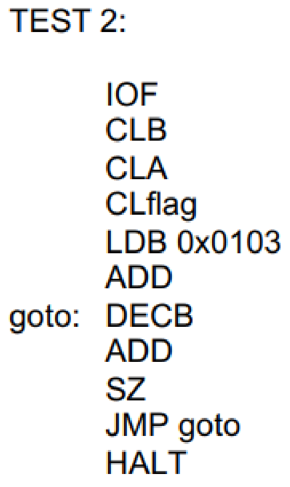
```
IOF
CLB
CLA
LDB 0x0104
LDA 0x0102
CMB
INCB
ADD
LDB 0x103
AND
STA 0x0500
LDB 0x0500
HALT
```



| Name | Value | Waveform |
|---|---|---|
| clk | 1 | |
| [2][15:0] | 0003 | 0002, 0000, ffff, 0000, 0001, 0003 |
| [1][15:0] | 0003 | 0, 0000, 0002, 0003 |

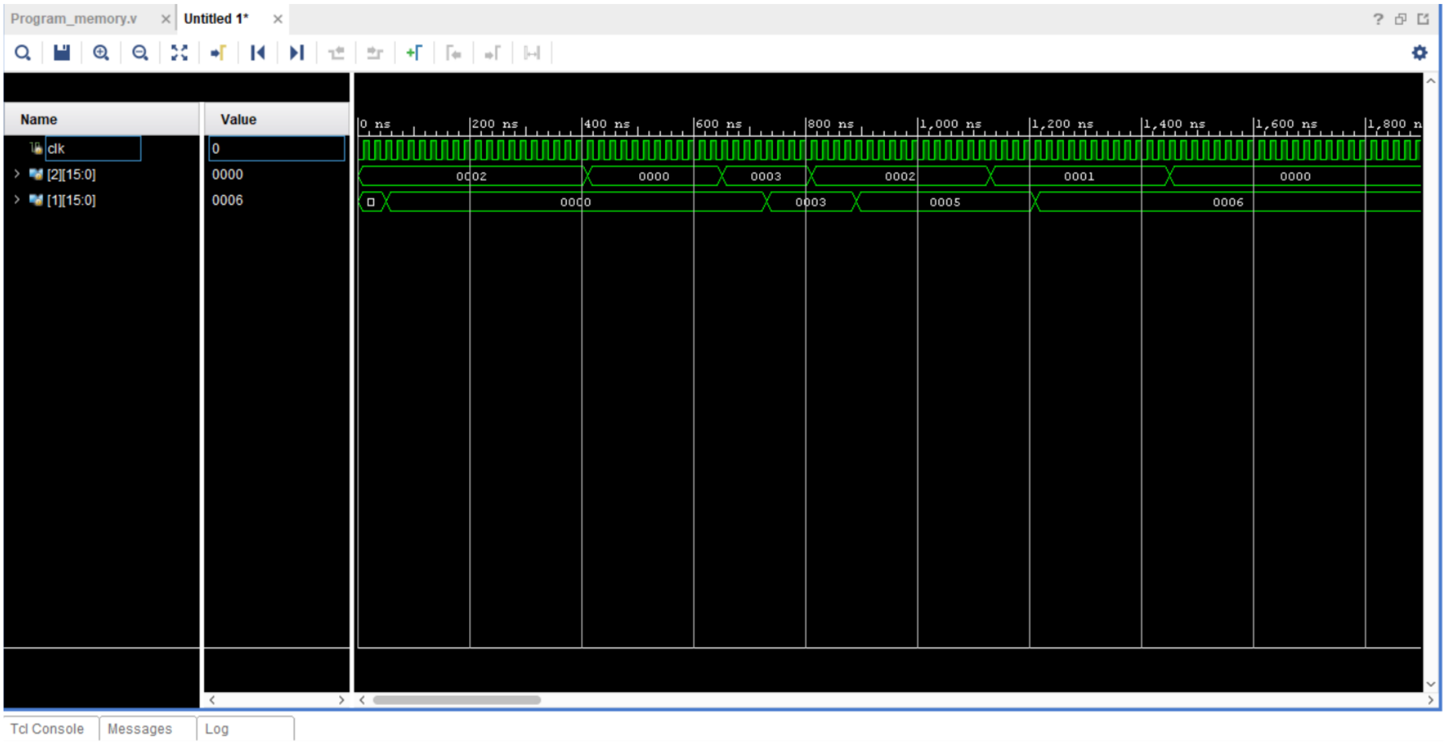
RESULT: A=3 ; B=3;

TEST_2:



TEST 2:

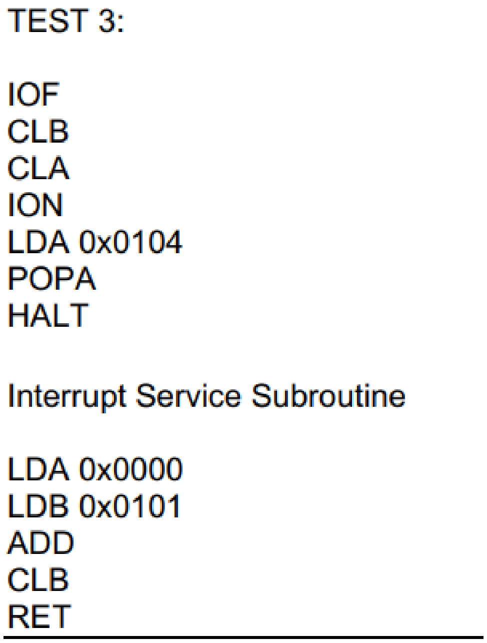
```
        IOF
        CLB
        CLA
        CLflag
        LDB 0x0103
        ADD
goto:   DECB
        ADD
        SZ
        JMP goto
        HALT
```



| Name | Value |
|---|---|
| clk | 0 |
| [2][15:0] | 0000 |
| [1][15:0] | 0006 |

RESULT: A=6; B=0;

TEST_3: TESTING the interrupt represented as IEN

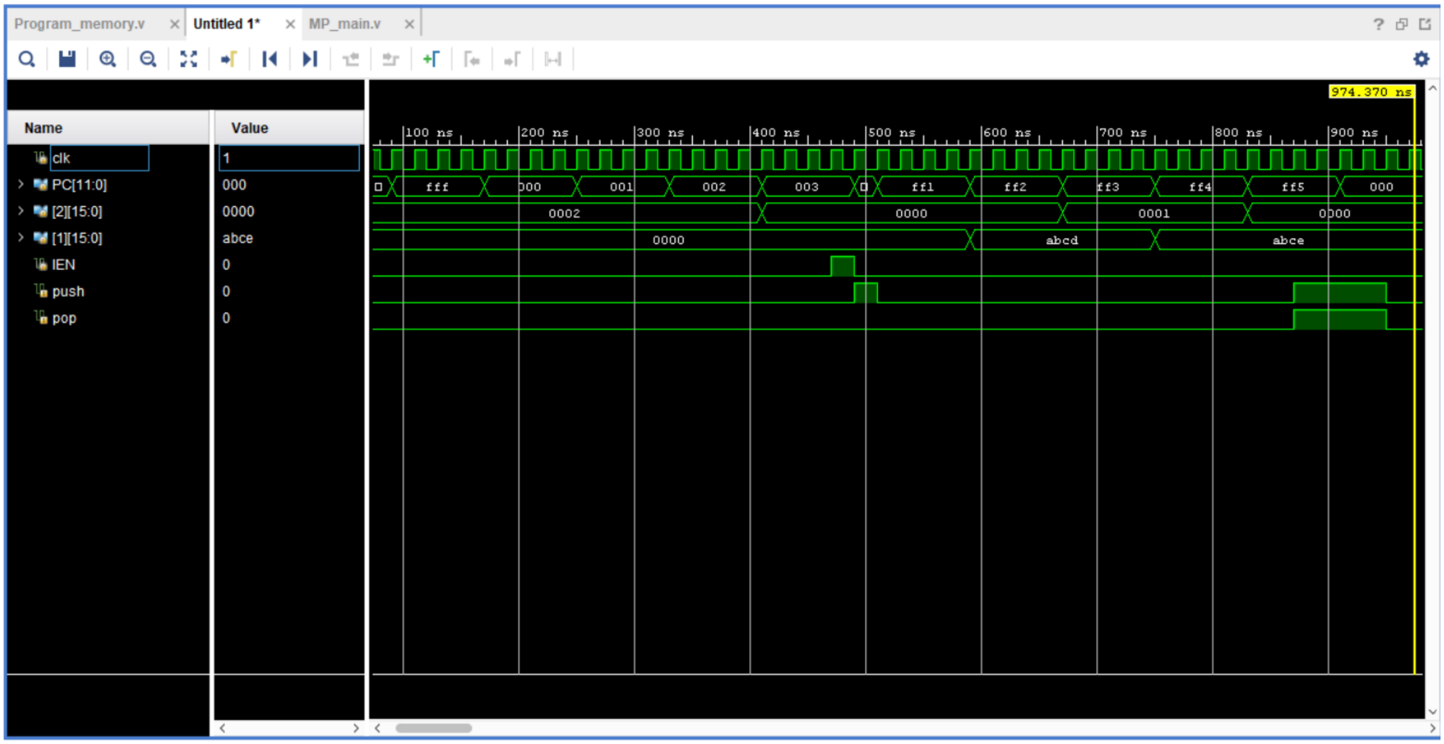


TEST 3:

```
IOF
CLB
CLA
ION
LDA 0x0104
POPA
HALT
```

Interrupt Service Subroutine

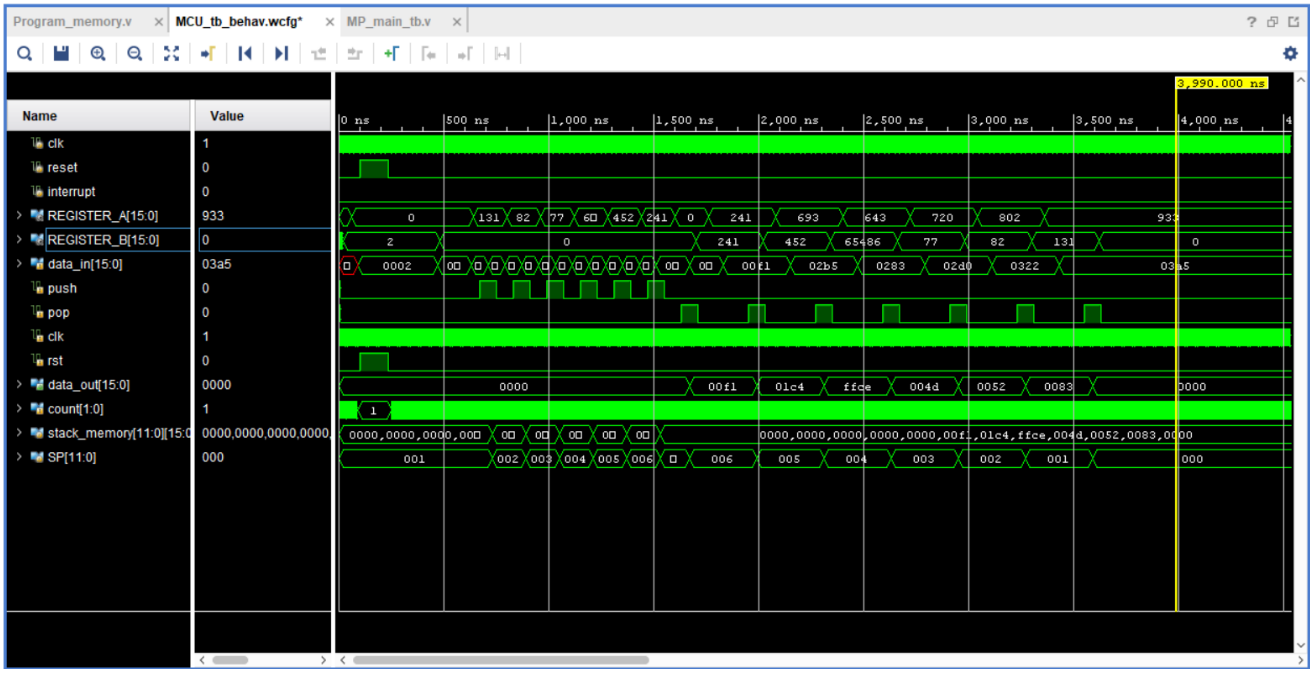
```
LDA 0x0000
LDB 0x0101
ADD
CLB
RET
```



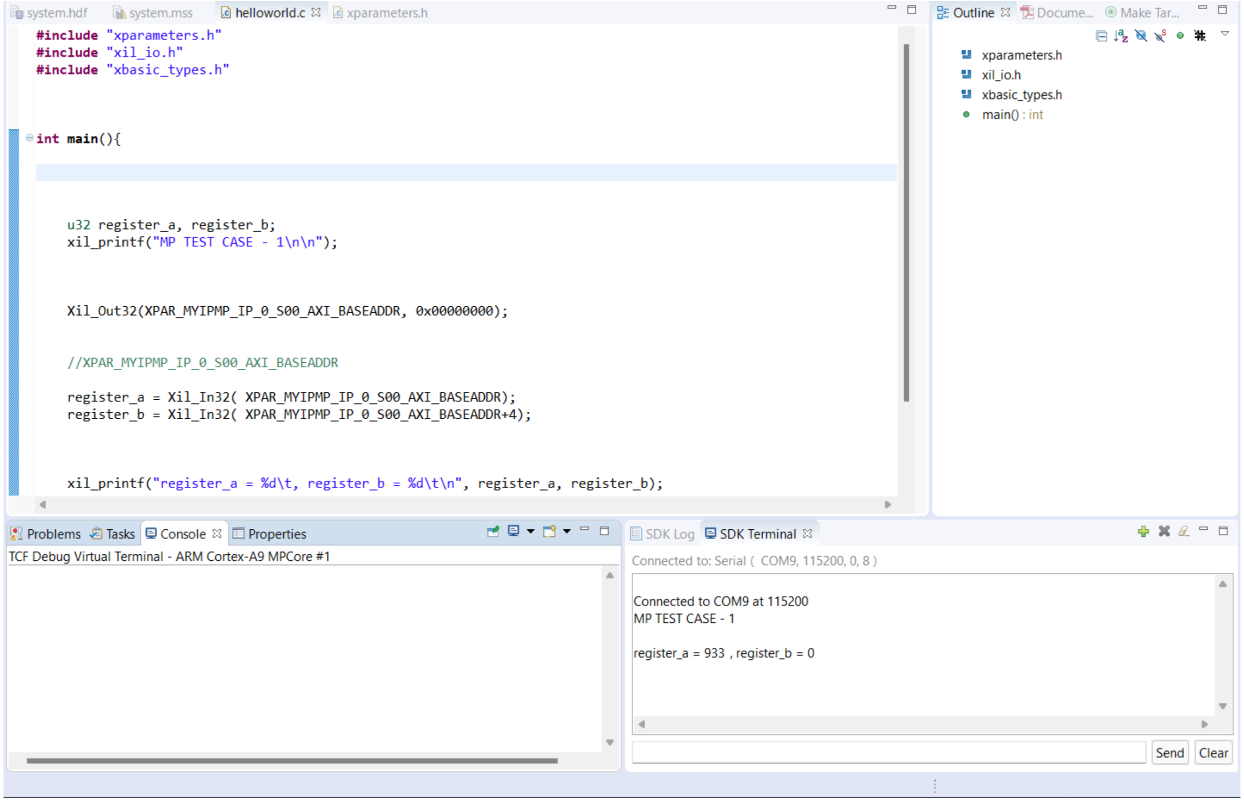
RESULT: A= 0xABCE; B=0;

## The Final test results:

TEST_1



SDK:



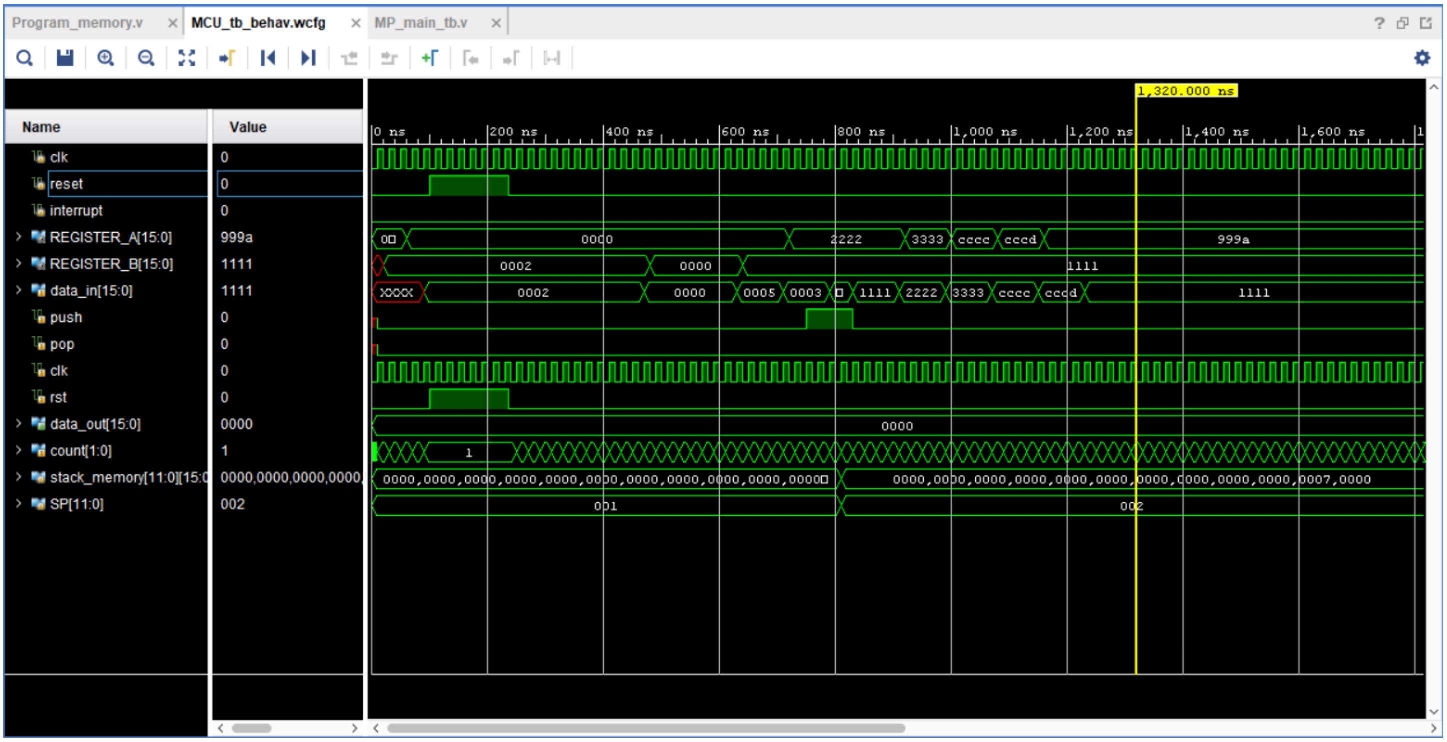
```
#include "xparameters.h"
#include "xil_io.h"
#include "xbasic_types.h"


int main(){



    u32 register_a, register_b;
    xil_printf("MP TEST CASE - 1\n\n");


    Xil_Out32(XPAR_MYIPMP_IP_0_S00_AXI_BASEADDR, 0x00000000);


    //XPAR_MYIPMP_IP_0_S00_AXI_BASEADDR

    register_a = Xil_In32( XPAR_MYIPMP_IP_0_S00_AXI_BASEADDR);
    register_b = Xil_In32( XPAR_MYIPMP_IP_0_S00_AXI_BASEADDR+4);


    xil_printf("register_a = %d\t, register_b = %d\t\n", register_a, register_b);
```

TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1

Connected to: Serial ( COM9, 115200, 0, 8 )

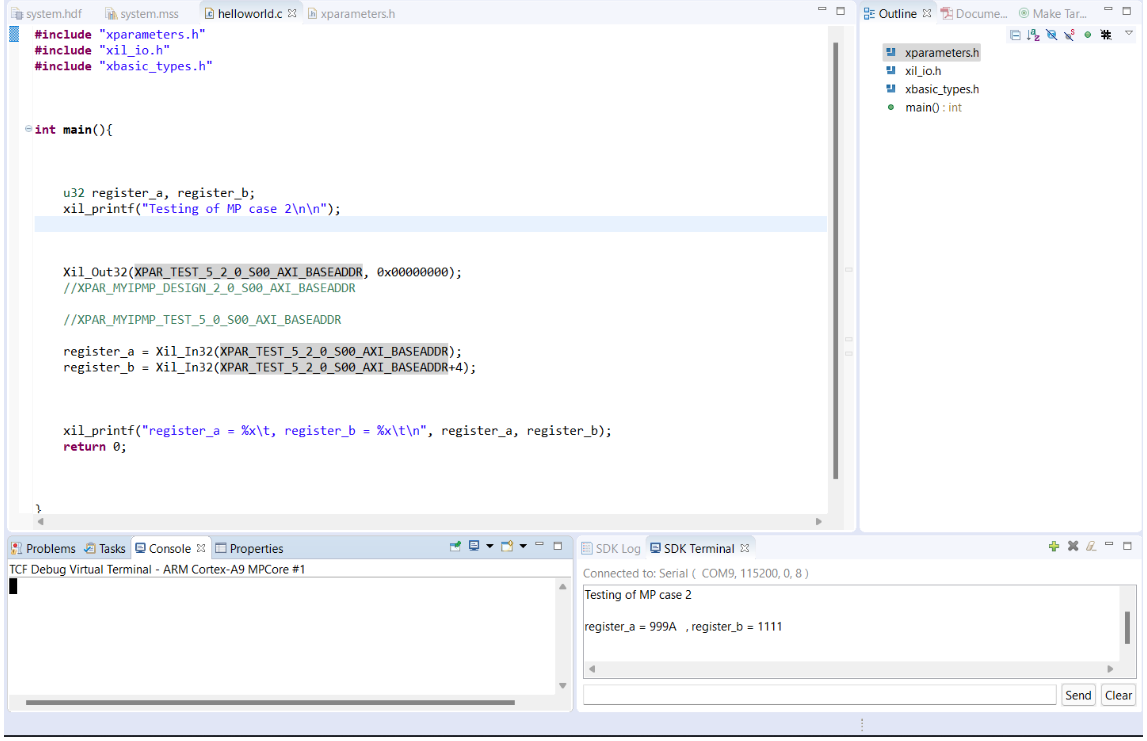
```
Connected to COM9 at 115200
MP TEST CASE - 1

register_a = 933 , register_b = 0
```

# TEST_2:



| Name | Value |
|---|---|
| clk | 0 |
| reset | 0 |
| interrupt | 0 |
| REGISTER_A[15:0] | 999a |
| REGISTER_B[15:0] | 1111 |
| data_in[15:0] | 1111 |
| push | 0 |
| pop | 0 |
| clk | 0 |
| rst | 0 |
| data_out[15:0] | 0000 |
| count[1:0] | 1 |
| stack_memory[11:0][15:0] | 0000,0000,0000,0000, |
| SP[11:0] | 002 |

# SDK:



```
#include "xparameters.h"
#include "xil_io.h"
#include "xbasic_types.h"


int main(){


    u32 register_a, register_b;
    xil_printf("Testing of MP case 2\n\n");



    Xil_Out32(XPAR_TEST_5_2_0_S00_AXI_BASEADDR, 0x00000000);
    //XPAR_MYIPMP_DESIGN_2_0_S00_AXI_BASEADDR

    //XPAR_MYIPMP_TEST_5_0_S00_AXI_BASEADDR

    register_a = Xil_In32(XPAR_TEST_5_2_0_S00_AXI_BASEADDR);
    register_b = Xil_In32(XPAR_TEST_5_2_0_S00_AXI_BASEADDR+4);



    xil_printf("register_a = %x\t, register_b = %x\t\n", register_a, register_b);
    return 0;


}
```

TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1

Connected to: Serial ( COM9, 115200, 0, 8 )

```
Testing of MP case 2

register_a = 999A   , register_b = 1111
```