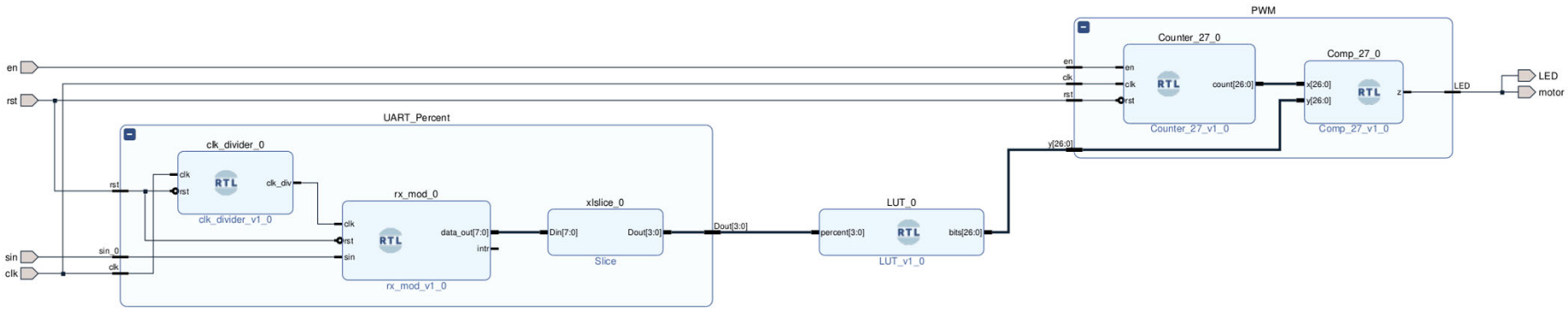
Memories and PS-PL Communication (ROM, RAM, BRAM, and AXI)

Emad Samuel Malki Ebeid

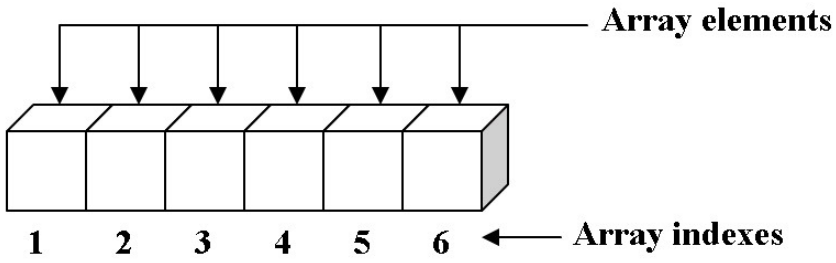
Lecture 2 summary



ROM

Array in VHDL

- Array has an index whose value selects each element.
- The index range determines:
  - Number of elements are in the array and their ordering (low *to* high, or high *downto* low).
- An index can be of any **integer** type.
- Array elements can be of **any** type.

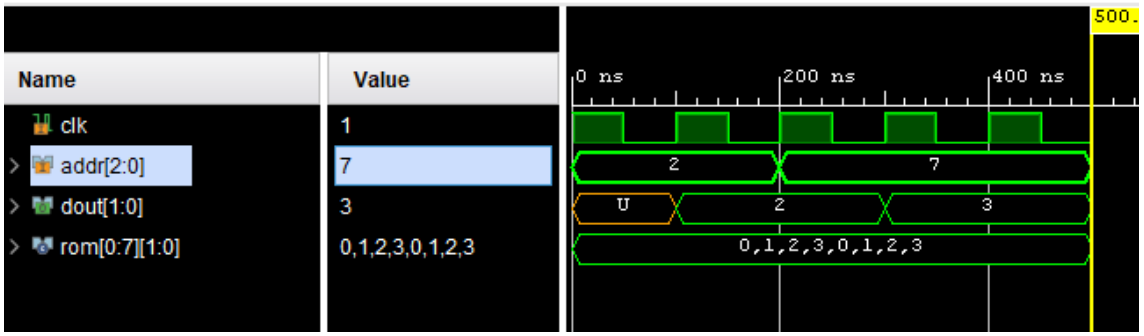


One-dimensional array with six elements

ROM

8x2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;


entity ROM_8_2 is
    Port ( clk  : in STD_LOGIC;
           addr : in std_logic_vector(2 downto 0);
           dout : out std_logic_vector(1 downto 0)
    );
end ROM_8_2;

architecture Behavioral of ROM_8_2 is
type rom_type is array (0 to 7) of std_logic_vector(1 downto 0);
constant rom : rom_type:=("00","01","10","11","00","01","10","11");
BEGIN
    process(clk)
    begin
        if(rising_edge(clk))then
        dout<=rom(conv_integer(unsigned(addr)));
    end if;
end process;
end Behavioral;
```

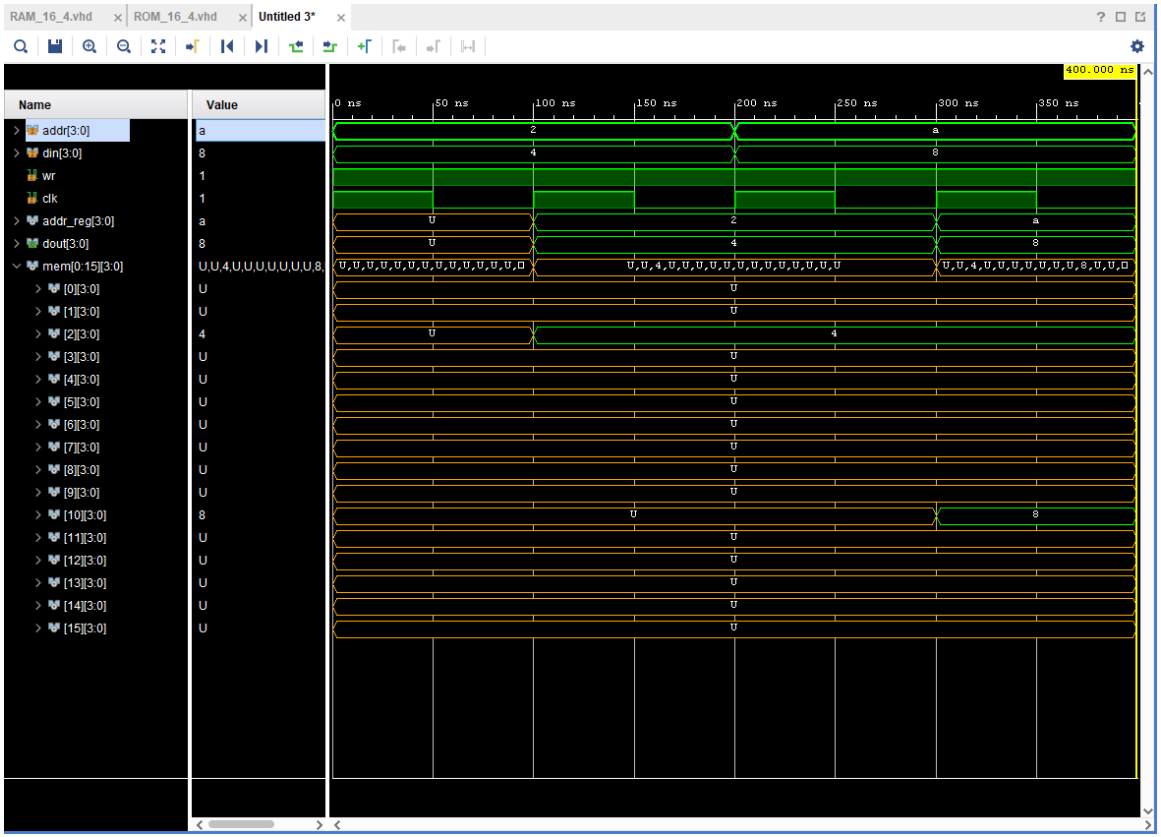


RAM

RAM

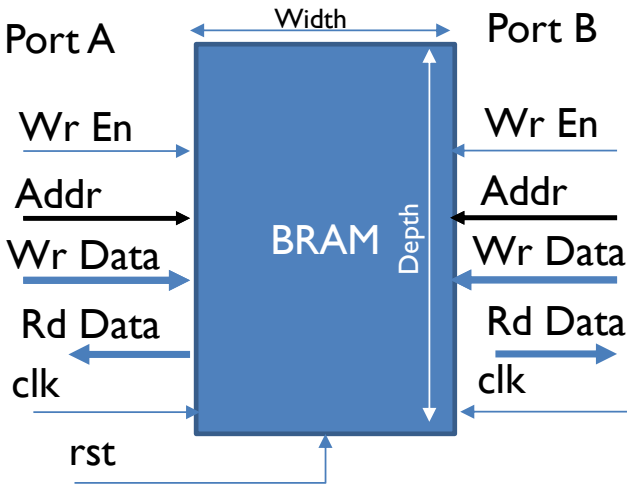
16x4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

entity RAM_16_4 is
    port(
          addr : in std_logic_vector(3 downto 0);
          din : in std_logic_vector(3 downto 0);
        wr,clk: in std_logic;
         dout :out std_logic_vector(3 downto 0)
         );
end RAM_16_4;

architecture Behavioral of RAM_16_4 is
type mem_type is array (0 to 15) of std_logic_vector(3 downto 0);
signal mem : mem_type;
signal addr_reg:std_logic_vector(3 downto 0);
BEGIN
    process(clk)
    begin
        if(rising_edge(clk))then
            if(wr='1')then
            mem(conv_integer(unsigned(addr)))<=din;
            end if;
        addr_reg<=addr;
        end if;
end process;
dout<=mem(conv_integer(unsigned(addr_reg)));

end Behavioral;
```



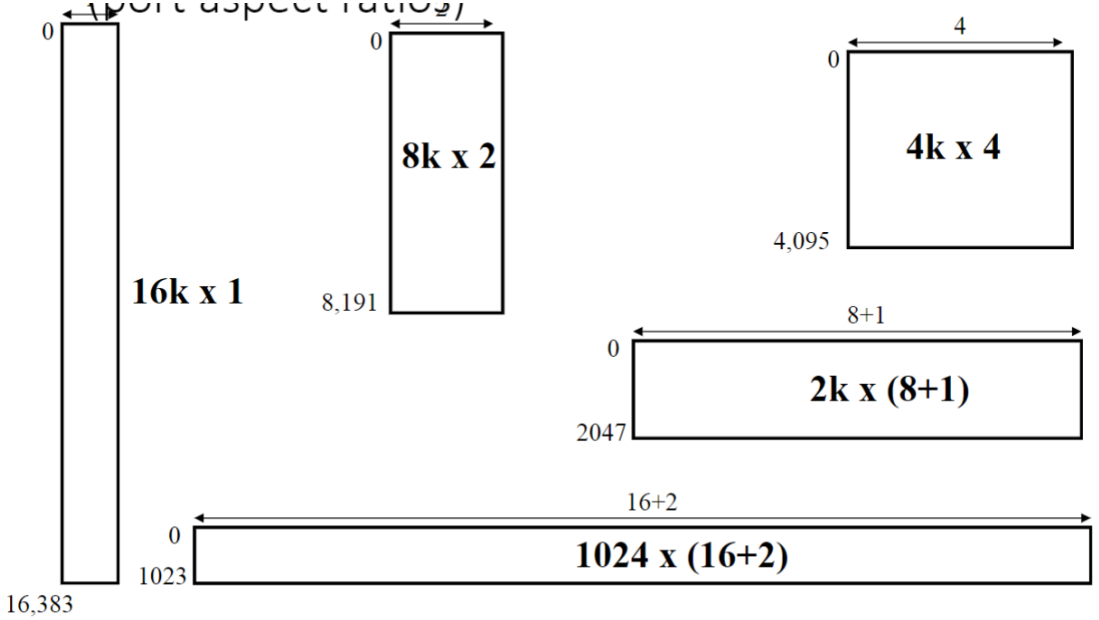
Block RAM

- Storing large amount of data (e.g., sensor data).
- Storing Read-only Data (calibration parameters).
- BRAM is a discrete parts of the FPGA.
- Each FPGA chip has a limited number of BRAM (Size=width*depth).
- Synchronous piece of logic
- Different configuration:
  - Single port, Dual port, FIFO



Wr En: Write Enable (1: write, 0: read)
Wr Data. Write data
Rd Data: Read Data

Block RAM configurations



Block RAM

Some of the key features of the block RAM include:

- Dual-port 36 Kb block RAM with port widths of up to 72
- Programmable FIFO logic
- Built-in optional error correction circuitry

Each device in the Zynq-7000 family has up to 755 dual-port block RAMs, each storing 36 Kb. Each block RAM has two completely independent ports that share nothing but the stored data.

Synchronous Operation

Each memory access, read or write, is controlled by the clock. All inputs, data, address, clock enables, and write enables are registered. The input address is always clocked, retaining data until the next operation. An optional output data pipeline register allows higher clock rates at the cost of an extra cycle of latency.

During a write operation, the data output can reflect either the previously stored data, the newly written data, or can remain unchanged.

Programmable Data Width

Each port can be configured as 32K × 1, 16K × 2, 8K × 4, 4K × 9 (or 8), 2K × 18 (or 16), 1K × 36 (or 32), or 512 × 72 (or 64). The two ports can have different aspect ratios without any constraints.

Each block RAM can be divided into two completely independent 18 Kb block RAMs that can each be configured to any aspect ratio from 16K × 1 to 512 × 36. Everything described previously for the full 36 Kb block RAM also applies to each of the smaller 18 Kb block RAMs.

Only in simple dual-port (SDP) mode can data widths of greater than 18 bits (18 Kb RAM) or 36 bits (36 Kb RAM) be accessed. In this mode, one port is dedicated to read operation, the other to write operation. In SDP mode, one side (read or write) can be variable, while the other is fixed to 32/36 or 64/72.

Both sides of the dual-port 36 Kb RAM can be of variable width.

Two adjacent 36 Kb block RAMs can be configured as one cascaded 64K × 1 dual-port RAM without any additional logic.

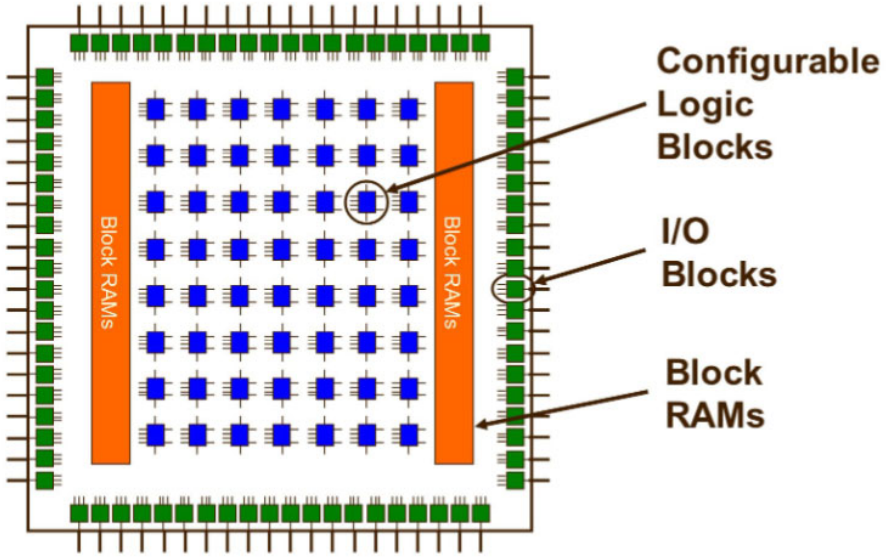
Error Detection and Correction

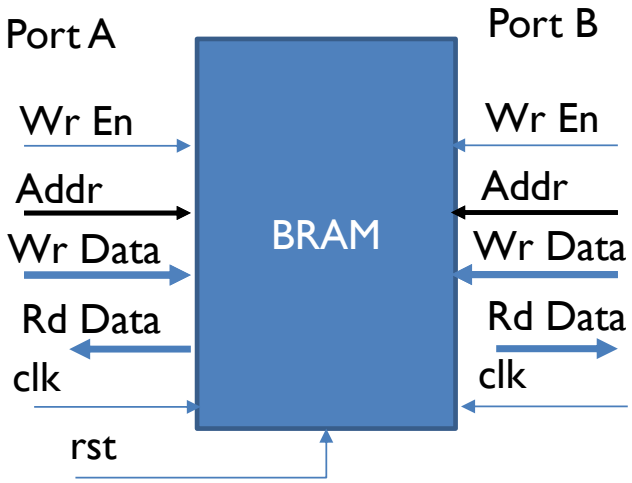
Each 64-bit-wide block RAM can generate, store, and utilize eight additional Hamming code bits and perform single-bit error correction and double-bit error detection (ECC) during the read process. The ECC logic can also be used when writing to or reading from external 64- to 72-bit-wide memories.

https://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf

Dual port BRAM configuration

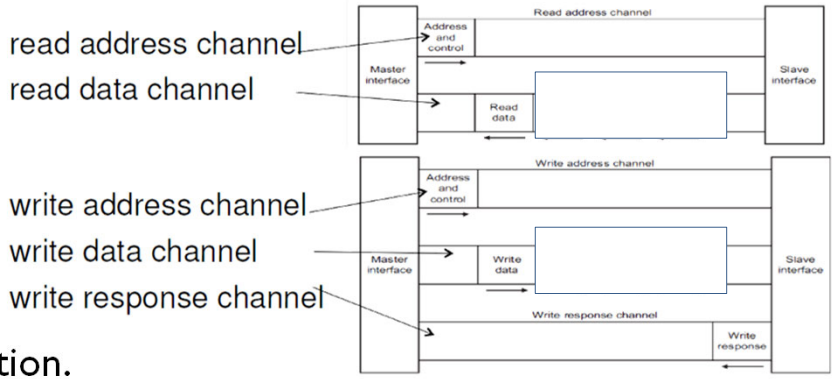
- Each port can run in different clock frequency
- One operation at a time (read or write at the same address)!





AXI protocol

- The Advanced eXtensible Interface (AXI), is part of ARM AMBA, a family of micro controller buses first introduced in 1996.
- It's a communication interface to interact with a RAM
- Reads/writes can have variable latencies
  - Write address can be sent at different times.
- Reads/writes can fail
- There are three types of AXI4 interfaces:
  - AXI4: for high-performance memory-mapped requirements.
  - AXI4-Lite: for simple, low-throughput memory-mapped communication.
  - AXI4-Stream: for high-speed streaming data.
- Data can move in both directions between the master and slave simultaneously, and data transfer sizes can vary.
- The limit in AXI4 is a burst transaction of up to 256 data transfers.
- AXI4-Lite allows only **one data transfer per transaction**.

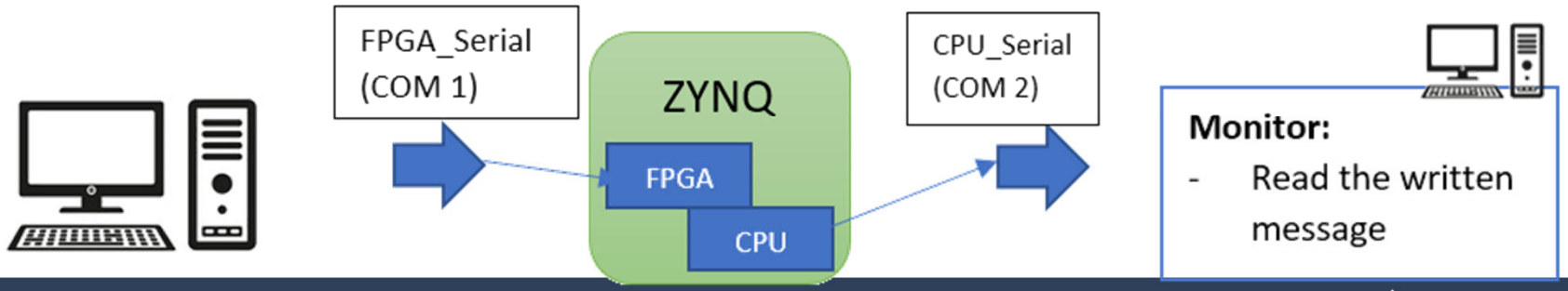


https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

Assignment 1

Assignment 1
Submission: 05-10-2024 23:59
(details in The submission page/ItsLearning)

- Design a digital circuit that stores a message coming from a PL UART entry and reads it through PS UART.
- The report should include the overall design (Vivado/Vitis snapshoots), simulation (PL), and execution (PS) outcomes. Make a video of the final implementation showing the testing process. Comment on the final testing results.
- ItsLearning/Resources,/Assignment 1
- Individual work, individual submission



For the next lecture

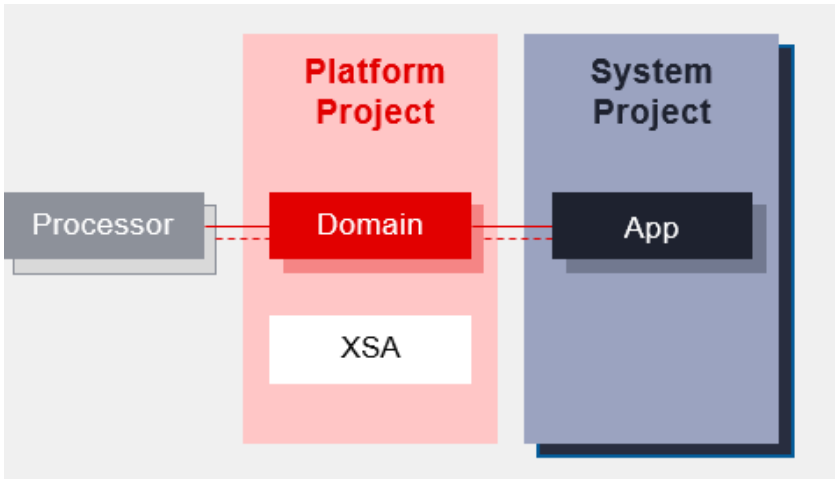
Remember to borrow the Ultra96v2 box from the library!

*The box has two motors and extra connections. That is for the final project work.

Lab Work

Vitis platform (Software Development)

- Create the application, then the platform, and import the XSA a file (generated by Vivado).
  - Built platform project
- Create application project



XSA

Xilinx® Support Archive (`.xsa`) is a Xilinx proprietary file format and only Xilinx software tools understand it. Third-party software tools can communicate to the XSCT Tcl interface to extract data from the `.xsa` file.

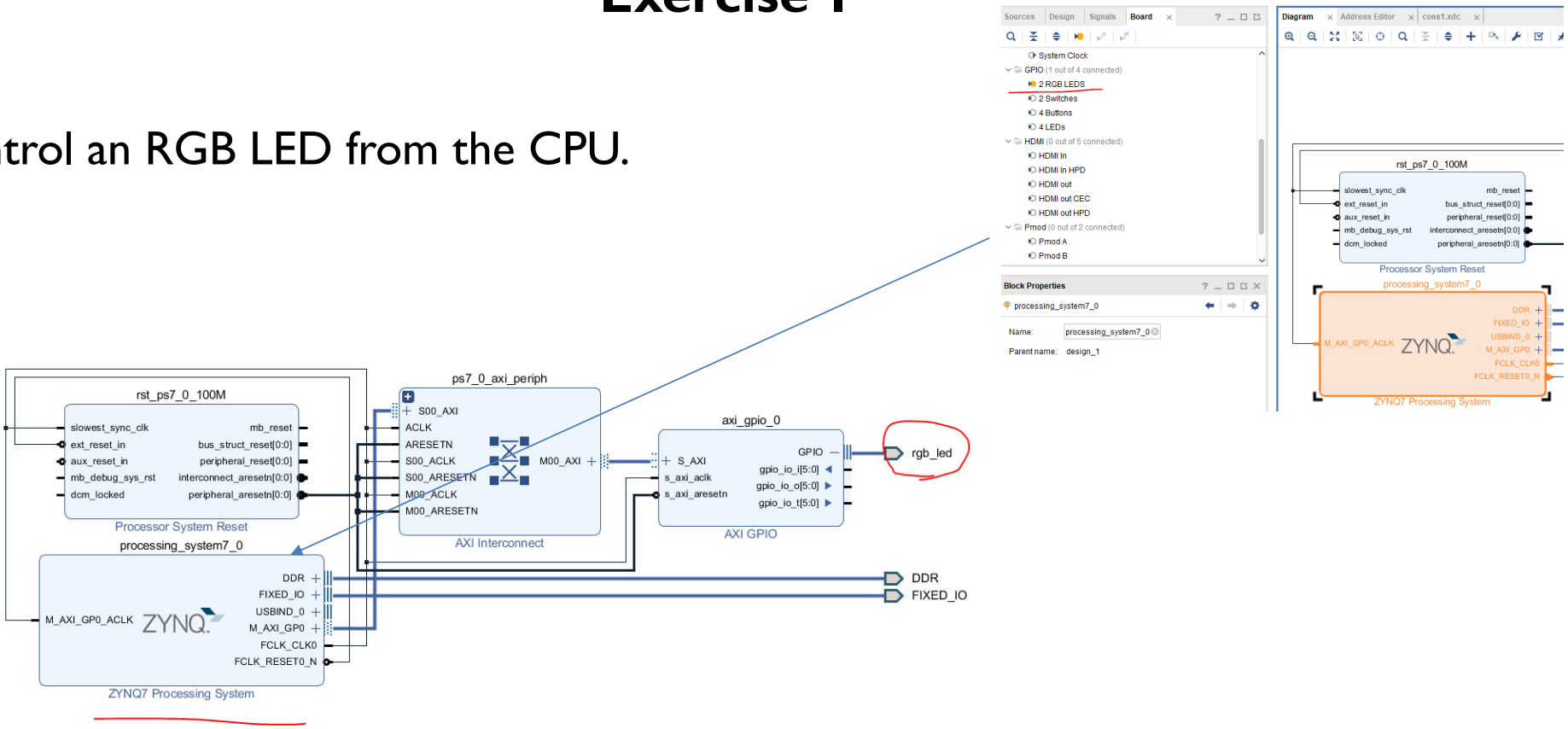
Note: Xilinx does not recommend manually editing the XSA file or altering its contents.

XSA is a container and contains:

- One or more `.hwh` files
  - Vivado® tool version, part, and board tag information
  - IP - instance, name, VLNV, and parameters
  - Memory Map information of the processors
  - Internal Connectivity information (including interrupts, clocks, etc.) and external ports information
- BMM/MMI and BIT files
- User and HLS driver files
- Other meta-data files

Exercise 1

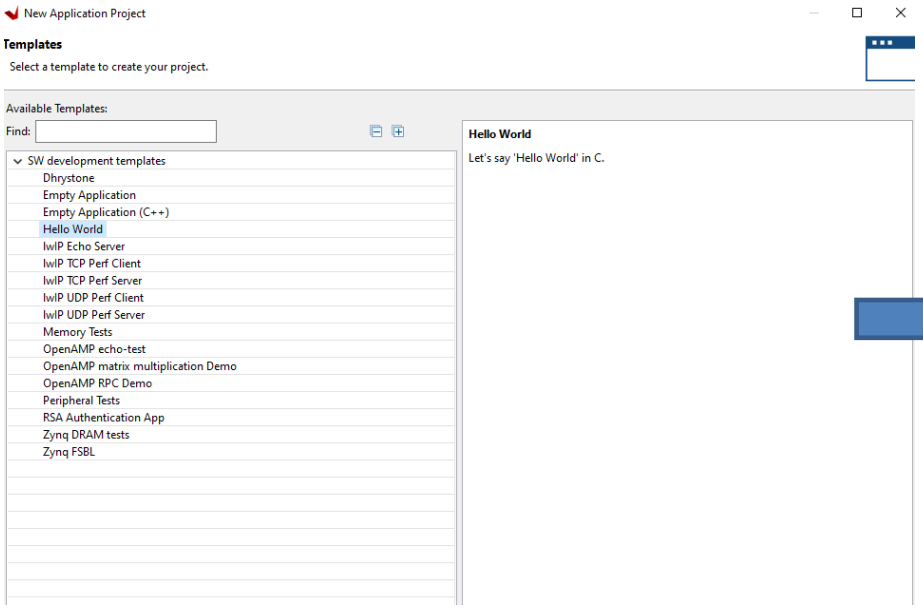
- Control an RGB LED from the CPU.



- The C code for triggering the LED is in the lecture folder

Hello World

- Interact with the ZYNQ board using the serial port.
- Write ”Hello World” in a C code and execute it in the processor and see the massage flowing back to the PC.



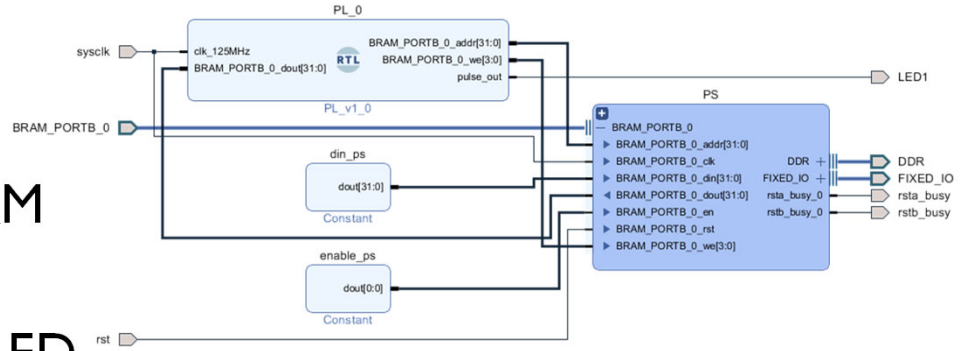

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"

int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    cleanup_platform();
    return 0;
}
```

LED GPIO

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xgpio.h"


#define LED 0b1111  /* Four LEDs Example LED_1: => 0001 */
XGpio Gpio; /* The Instance of the GPIO Driver */

int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    int status;
        status = XGpio_Initialize(&Gpio, XPAR_AXI_GPIO_0_DEVICE_ID);
        if (status != XST_SUCCESS) {
            print("Err: Gpio initalization failed\n\r");
        } else {
            print("Info: Gpio Initialization successful\n\r" );
        }

        XGpio_SetDataDirection(&Gpio, 1, 0); /* define which AXI channel is used with the LEDs (1 or 2) and the direction (LED is output '0') */

        while (1) {

                XGpio_DiscreteWrite(&Gpio, 1, LED); /* Write the previous parameters to the GPIO */
            }
    cleanup_platform();
    return 0;
}
```

Exercise 2: BRAM

- Design a digital circuit to interface PS with PL utilizing the BRAM.
- Write a VHDL code to access PortB of the BRAM and use the value to switch ON/OFF and LED.
- Write a C code to assign a value to the BRAM LED Address to switch it ON/OFF.



Outcome:

- Know how to assign a value to a BRAM address.
- Know how to read a BRAM value from the PL.

!BRAM: Port Aspect Ratio

- The BRAM memory address needs to be incremented by 4, not by 1 to move to the next 32-bit word in memory (Please read Xilinx BRAM manual*, page 48.)
- In the exercise, PortB addresses (PL side) needs to align with the PortA (PS side) (e.g., [0, 4, 8, 12, ...].)

Consider a True Dual-port RAM of 32x2048, which is the A port width and depth. From the perspective of an 8-bit B port, the depth would be 8192. The `addra` bus is 11 bits, while the `addrb` bus is 13 bits. The data is stored little-endian, as shown in Figure 3-12. Note that A_n is the data word at address *n*, with respect to the A port. B_n is the data word at address n with respect to the B port. A_0 is comprised of B_3, B_2, B_1, and B_0.

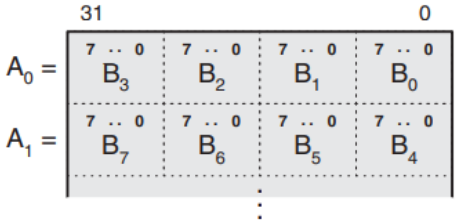
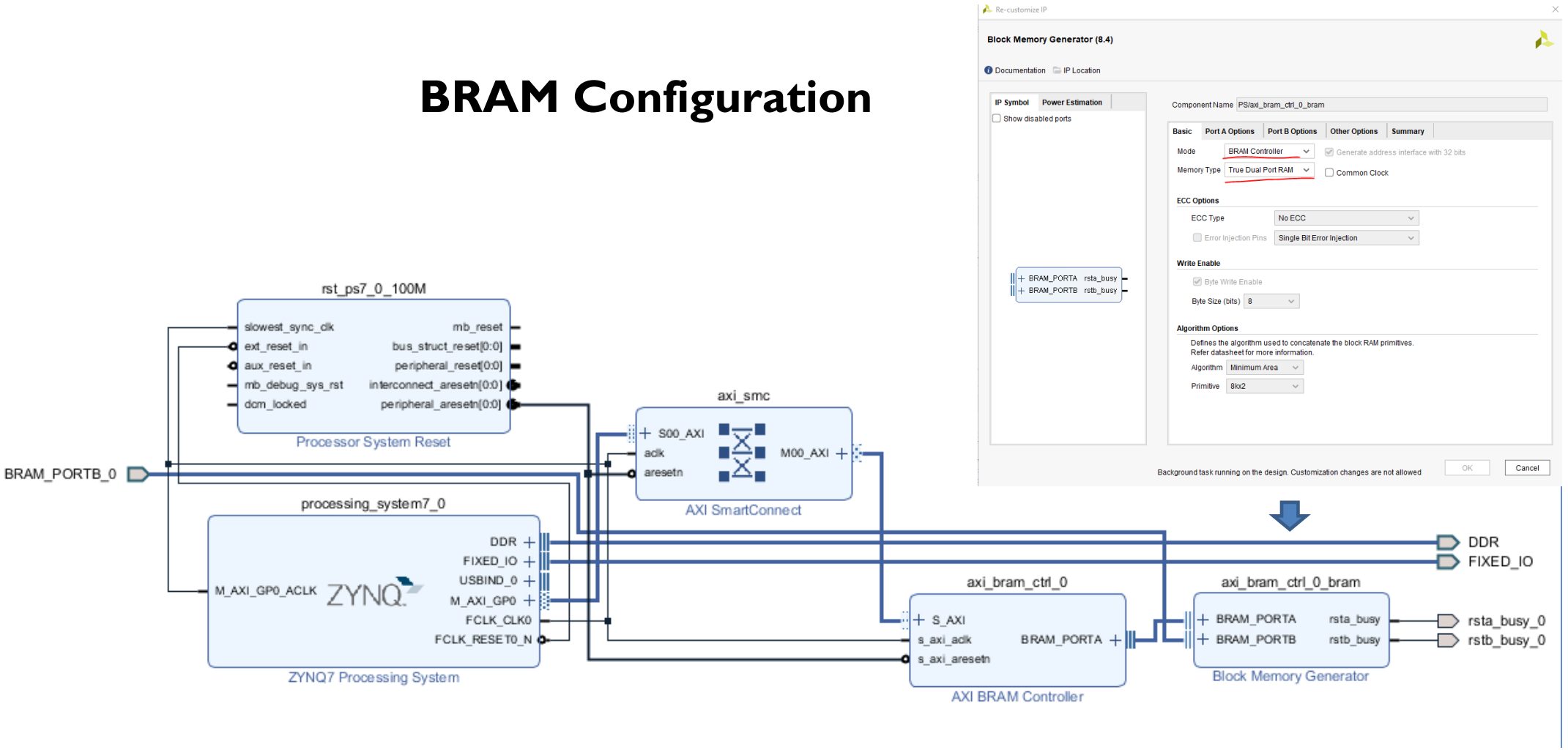


Figure 3-12: **Port Aspect Ratio Example Memory Map**

* https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf

- https://www.embedded.com/endianness/

BRAM Configuration



BRAM PS Code

- Code is in the lecture folder

```
/*
 * BRAM_LED.c: simple BRAM_LED test application
 *
 * By: Emad Samuel Malki Ebeid
 * email: esme@mmmi.sdu.dk
 *
 * ----------------------------------------------------
 * | UART TYPE   BAUD RATE                            |
 * ----------------------------------------------------
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xbram.h"
#include "sleep.h"

#define MYMEM_u(A) ((volatile u32*)ConfigPtr->MemBaseAddress)[A]
//#define MAXMEM       ((ConfigPtr->MemHighAddress-ConfigPtr->MemBaseAddress)+1)/sizeof(u32)

XBram Bram;
XBram_Config *ConfigPtr;
int initBRAM();

int main()
{
    init_platform();

    initBRAM();

    xil_printf("BRAM initialisation complete.\r\n");

    int mem_value, addr_value=1;

  while(1)
  {
        xil_printf("Enter memory value for address %d:\r\n", addr_value);
        scanf("%d", &mem_value);

        MYMEM_u(addr_value)=mem_value;              //corresponds to memory address 4 on the FPGA (X"40000004")

        xil_printf("Memory value %d for address %d updated successfully.\r\n", mem_value, addr_value);

     usleep(1000);

  } //end while loop

    cleanup_platform();
    return 0;
}


/*
 * This function initialises the BRAM driver. If an error occurs then exit
 */

int initBRAM()
{

    //BRAM initialization
    xil_printf("Initialising block RAM...\n\r");
    int Status;

    ConfigPtr = XBram_LookupConfig(XPAR_BRAM_0_DEVICE_ID);
    if (ConfigPtr == (XBram_Config *) NULL) {
        return XST_FAILURE;
    }

    Status = XBram_CfgInitialize(&Bram, ConfigPtr,
            ConfigPtr->CtrlBaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    xil_printf("Done.\r\n");
    return XST_SUCCESS;
}
```