Hands-On 4: AXI BRAM

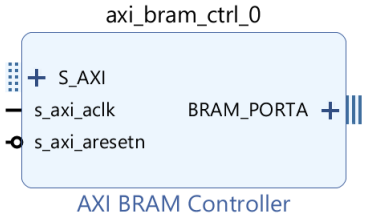
Objective

- The goal of this hands-on exercise is to get familiar with the AXI BRAM module from Xilinx
  - Part 1: Create a Vivado project with AXI BRAM interfaces
  - Part 2: Create a Vitis project to update a shared dual-port memory

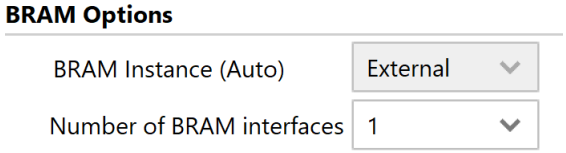
Part 1: Create a Vivado project with AXI BRAM interfaces

Vivado Project

- Start by opening the Vivado project developed in Hands-on 1: **HO1_HDMI**
- Save it as new project (File > Project > Save As) with name **H04_AXI_BRAM**



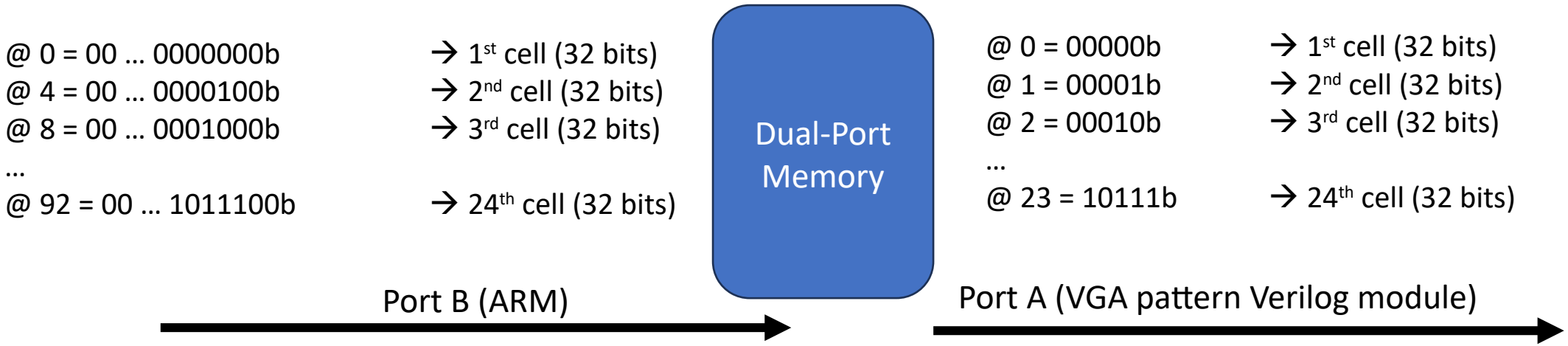
- On the block design, add an **AXI BRAM Controller** IP module
- Double-click to configure it. Change the number of BRAM interfaces to 1



- Expand the interface labeled **BRAM_PORTA**
- We will connect this to PortB of the dual-port memory containing the dinosaur image
- Make the following direct connections
  - bram_clk_a → iClkB [clock signal]
  - bram_wrdata_a → iDataB [data from the PS to the memory]
  - bram_rddata_a → oDataB [data from the memory to the PS]

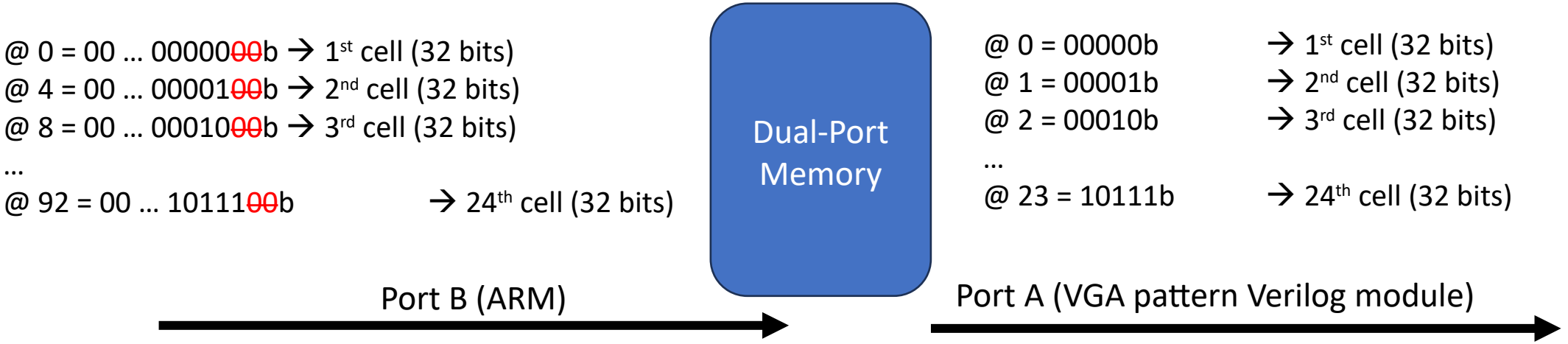
Vivado Project

- The next connection should be the address signal:
  - bram_addr_a → iAddrB [address signal]
- **But** if we make a direct connection, the design will not work
  - There is a discrepancy between the **memory addressing modes** of the ARM core and our Verilog module
- Our Verilog memory was defined with 24 addresses (cells), each with 32-bits
- ARM indexes memory addresses at byte-level, that is, there is 1 address per each byte
- To access 32-bit words from the ARM core, addresses are by definition a multiple of 4

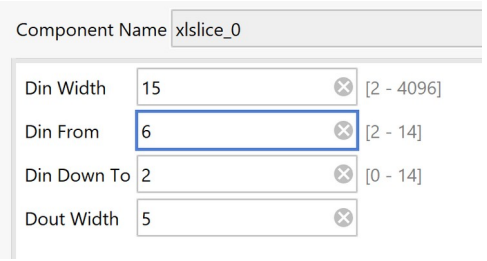


Vivado Project

- In order to fix this problem, we need to map the two address spaces
- This can be done by dropping the 2 lower bits of the ARM address – then they match!

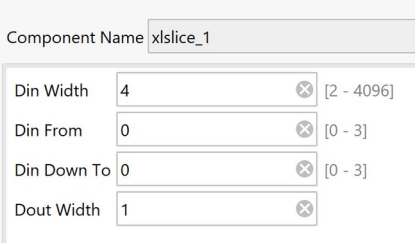


- To drop bits from an array in block design, you can use the **Slice** IP module from Xilinx
- Add it to the design, and configure it as follows:
  - From the 15 address bits provided by the BRAM controller, this module will keep only 5 bits (from bit 6 to bit 2), dropping bits 1 and 0

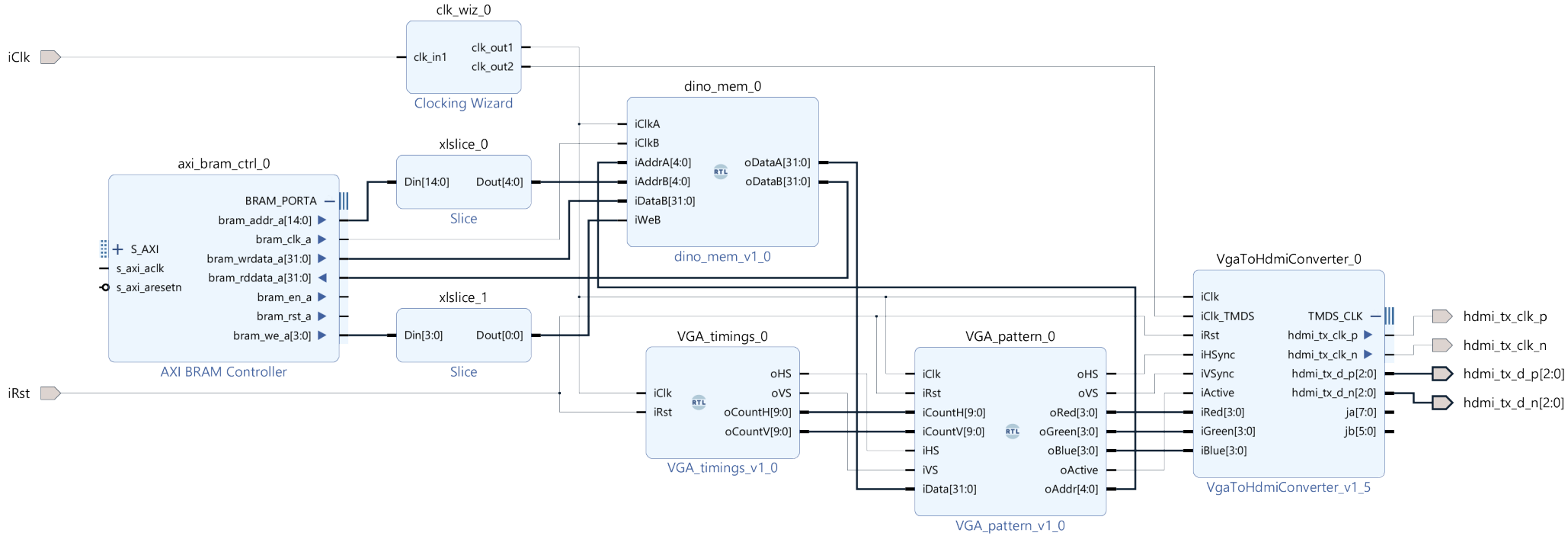


Vivado Project

- Since our memory is not byte addressable, we will also use a slice IP block to make the connection:
  - bram_we_a → iWeB [write enable signal]

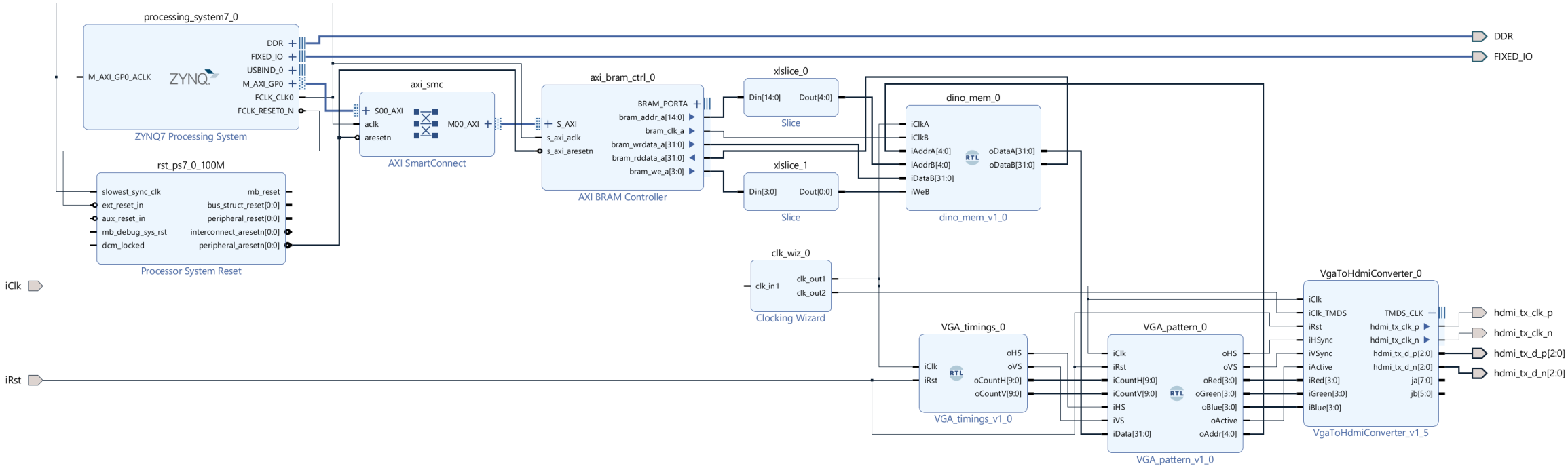


- Add another slice IP block, and configure it to take only bit 0 from the 4 bits
- Your edited design should look as shown below:



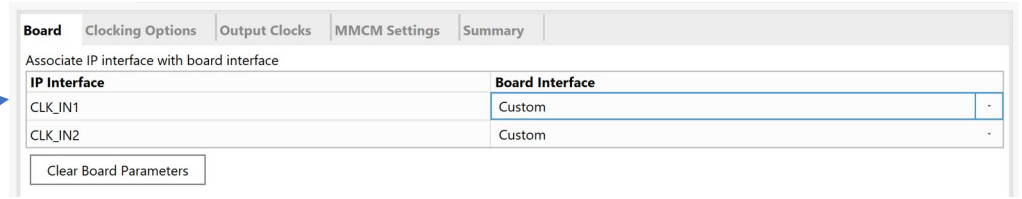
Vivado Project

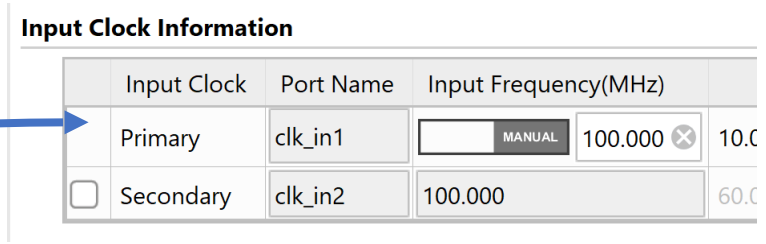
- Add now a **ZYNQ7 Processing System**, and click on **Run Block Automation**
- Click now on Run Connection Automation
- Vivado will add an **AXI Interconnect** between the M_AXI_GP0 port of the PS and the AXI BRAM



Vivado Project

- The original H01_HDMI design uses the external clock (125 MHz) to drive the Clock Wizard IP
- Since now we have a ZYNQ7 module, we can use an internal clock
  - And get rid of the Ethernet connection to get a stable clock
- Remove the *iClk* port from the block design
- Connect the FCLK_CLK0 to the input of the Clocking Wizard IP
- Double-click on the Clock Wizard and change CLK_IN1 from sys_clk back to **Custom**
- Under Clocking options, change also the clock frequency of clk_in from 125 MHz to **100 MHz**

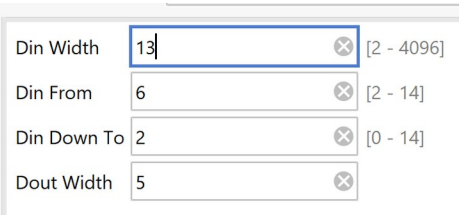


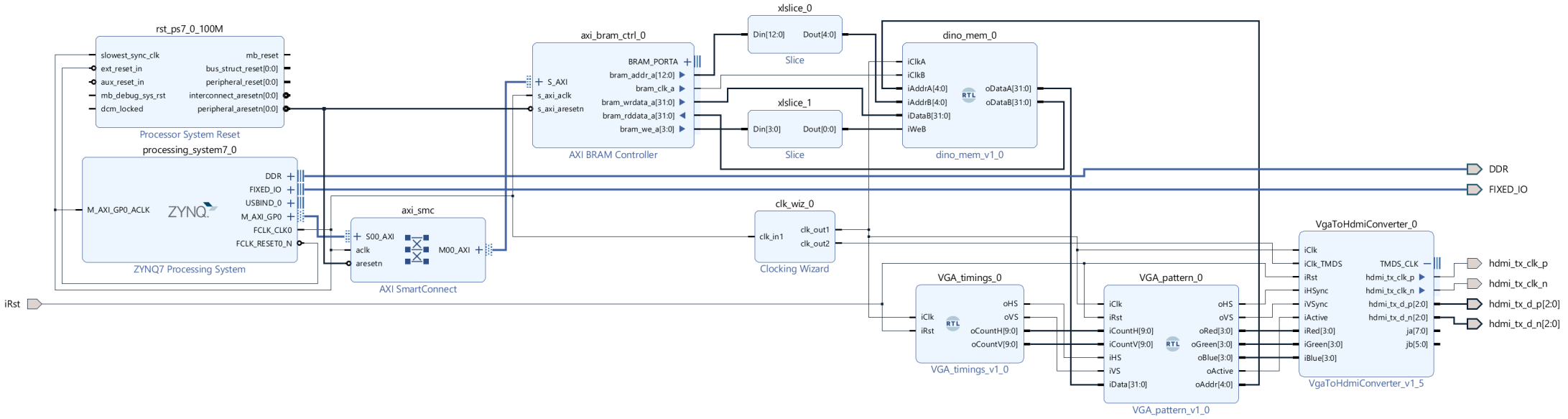


- **Comment lines 8 and 9 from the XDC constraints file, since we are not using anymore *iClk***

Vivado Project

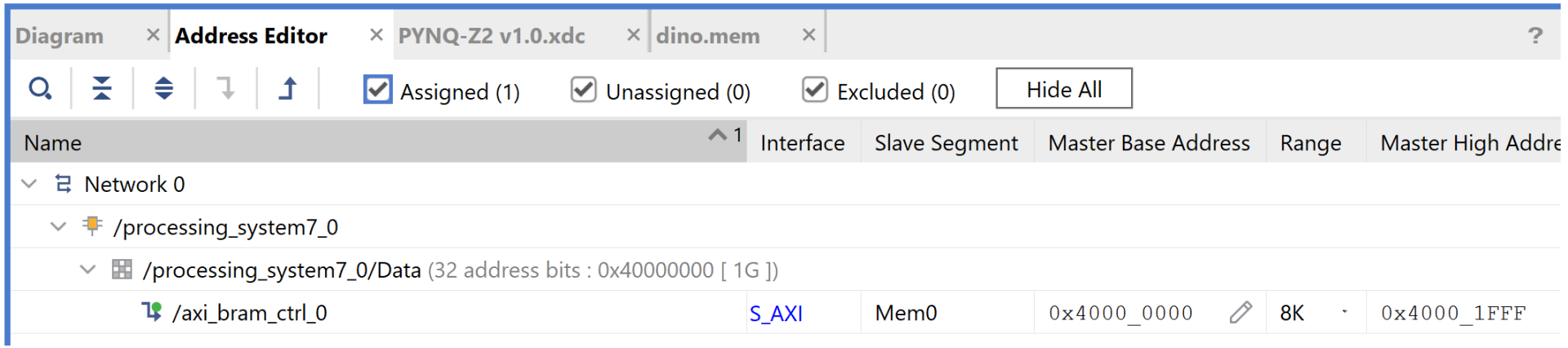
- Your final design should look as shown below
- **Important**: it is possible that the **bram_addr_a** of the AXI BRAM Controller reduces from 15 bits to 13 bits, when Vivado automates connections
  - Make sure to adapt the slice interface: change Din width from 15 to 13
- Click on the **Validate Design** to make sure there are no errors
- You can now **Generate Bitstream** and export the **XSA** hardware platform





Vivado Project

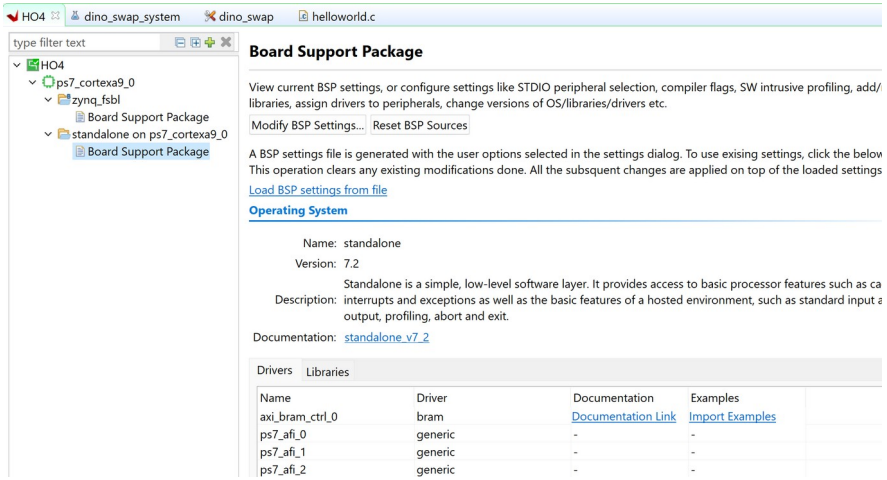
- You can see in the Address Editor that the (base) memory address assigned to the BRAM controller is 0x4000_0000
- In the next part, we will access the shared memory in Vitis through this address range



Part 2: Create a Vitis project to update a shared dual-port memory

Vitis Project

- Launch the **Vitis 2024.1** , select a path for the Vitis workspace (for instance **HO4_Workspace**)
- Click on **Create Platform Project**, give it the name HO4, import the XSA created in Vivado and build it
- Click on the **Board Support Package** to see the drivers associated to the platform
- We can see that Vitis has added software drivers to control our axi_bram_ctrl module
- Similar as before, you have from here access to **Documentation** as well as **Examples**



Vitis Project

- Download the file **dino_swap.c**
- Create a new **Hello World** application
- Replace the contents of the original *hello_world.c* file with the ones from *dino_swap.c*

- Build the downloaded application and program it in the board
- Connect the HDMI Out interface of the Pynq-Z2 to a monitor
- You should see the original image of the dinosaur updating every 250ms
- Why? The C code running on the ARM core is updating the shared memory contents

Vitis Project

Take a look at the code and make sure you understand how it works

- The dino swap file contains two 24x32-bit static arrays corresponding to dino images (in hexadecimal)
- The first array is the same image as originally stored in the memory (dino.mem file)
- The second array image is similar, but the dinosaur has the other leg up
- The code has a while() loop, which continuously updates the contents of the shared Verilog memory with one of the 2 images
- The usleep() function from **`sleep.h`** pauses the program execution for 250ms

```
const u32 dino_mem[2][24] =
{
    {
        0x00000000,0x0000FFE0,0x0001BFF0,0x0001FFF0,0x0001FFF0,0x0001FFF0,
        0x0001FFF0,0x0001FC00,0x0001FF80,0x0001F800,0x0803F000,0x080FF700,
        0x0C3FFD00,0x0E7FF800,0x0FFFF800,0x0FFFF800,0x07FFF000,0x03FFC000,
        0x01FF0000,0x00FE0000,0x00778000,0x00600000,0x00400000,0x00600000
    },
    {
        0x00000000,0x0000FFE0,0x0001BFF0,0x0001FFF0,0x0001FFF0,0x0001FFF0,
        0x0001FFF0,0x0001FC00,0x0001FF80,0x0001F800,0x0803F000,0x080FF700,
        0x0C3FFD00,0x0E7FF800,0x0FFFF800,0x0FFFF800,0x07FFF000,0x03FFC000,
        0x01FF0000,0x00FE0000,0x00660000,0x00320000,0x00020000,0x00030000
    }
};
```

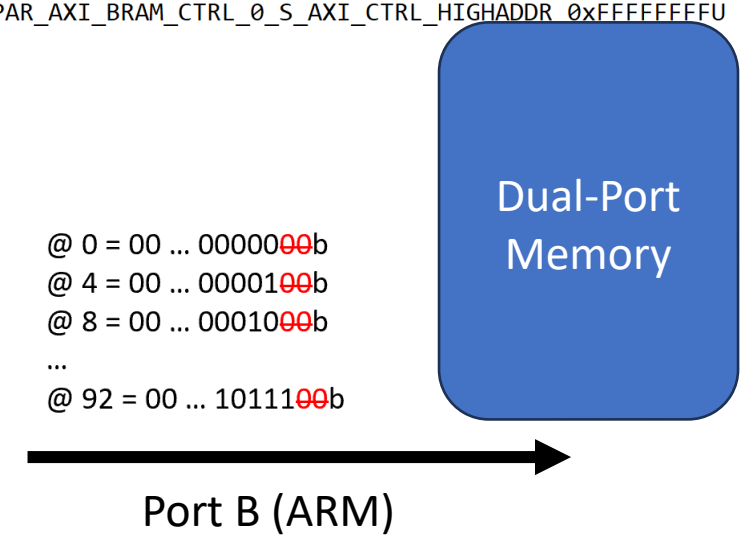
```
00000000000000000000000000000000 = 0x00000000
00000000000000001111111111100000 = 0x0000FFE0
00000000000000011011111111110000 = 0x0001BFF0
00000000000000011111111111110000 = 0x0001FFF0
00000000000000011111111111110000 = 0x0001FFF0
00000000000000011111111111110000 = 0x0001BFF0
00000000000000011111111111110000 = 0x0001BFF0
00000000000000011111110000000000 = 0x0001FC00
00000000000000011111111110000000 = 0x0001FF80
00000000000000011111110000000000
00001000000000111111000000000000
00001000000001111111011100000000
00001100001111111111110100000000
00001110011111111111100000000000
00001111111111111111100000000000
00001111111111111111100000000000
00000111111111111111000000000000
00000111111111111110000000000000
00000011111111111100000000000000
00000011111111111000000000000000
00000001111111100000000000000000
00000000111111100000000000000000
00000000011101111000000000000000
00000000011000000000000000000000
00000000010000000000000000000000
00000000011000000000000000000000
```

Vitis Project

- To access the memory, the code uses the **Xil_In32** (read) and **Xil_Out32** (write) functions
- These are defined in the **`xil_io.h`** library, and allow to read/write to memory-mapped addresses in the AXI address space
- In the **`xparameters.h`** file, you can see that the base address where read/write operations are performed on is XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR
- This corresponds to address 0x4000_0000, consistent with the Address Editor we saw in Vivado

```
33 /******************************************************************/
34 /* Definitions for driver BRAM */
35 #define XPAR_XBRAM_NUM_INSTANCES 1U
36
37 /* Definitions for peripheral AXI_BRAM_CTRL_0 */
38 #define XPAR_AXI_BRAM_CTRL_0_DEVICE_ID 0U
39 #define XPAR_AXI_BRAM_CTRL_0_DATA_WIDTH 32U
40 #define XPAR_AXI_BRAM_CTRL_0_ECC 0U
41 #define XPAR_AXI_BRAM_CTRL_0_FAULT_INJECT 0U
42 #define XPAR_AXI_BRAM_CTRL_0_CE_FAILING_REGISTERS 0U
43 #define XPAR_AXI_BRAM_CTRL_0_UE_FAILING_REGISTERS 0U
44 #define XPAR_AXI_BRAM_CTRL_0_ECC_STATUS_REGISTERS 0U
45 #define XPAR_AXI_BRAM_CTRL_0_CE_COUNTER_WIDTH 0U
46 #define XPAR_AXI_BRAM_CTRL_0_ECC_ONOFF_REGISTER 0U
47 #define XPAR_AXI_BRAM_CTRL_0_ECC_ONOFF_RESET_VALUE 0U
48 #define XPAR_AXI_BRAM_CTRL_0_WRITE_ACCESS 0U
49 #define XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR 0x40000000U
50 #define XPAR_AXI_BRAM_CTRL_0_S_AXI_HIGHADDR 0x40001FFFU
51 #define XPAR_AXI_BRAM_CTRL_0_S_AXI_CTRL_BASEADDR 0xFFFFFFFFU
52 #define XPAR_AXI_BRAM_CTRL_0_S_AXI_CTRL_HIGHADDR 0xFFFFFFFFU
```

- Note that the C code accesses always addresses in increments of 4
  - *Accessing 32-bit word memory elements with addresses not multiple of 4 (for instance: address 0x00…001) is not allowed !*
- This is why the removing of the 2 lower bits is handled internally by the slice module in the FPGA



Vitis Project

- For bare-metal applications, accessing AXI memory-mapped interfaces can always be done with the **Xil_In32** and **Xil_Out32** functions
- Even the previous AXI GPIO examples could also be solved with these two functions, instead of using the (more bloated) XGPIO driver library

- This hands-on example shows how to share memories between SW and HW components
- This is a very useful interface in HW/SW Co-Design

- Next week we will see how to access AXI memory space outside bare-metal applications, that is, when running an **Operating System**