# Main Memory Characterizer for PynqZ2

Valentino Guerrini

January 11, 2023

**Abstract**

The goal of this project is to design and implement a memory characterizer for the PYNQ-Z2 platform using Vitis HLS for the IP block design and Python to develop the host code. The memory characterizer will be able to analyze the performance characteristics of DDR memory modules of the PYNQ-Z2 board, such as access times and bandwidth during reading and writing. The IP block implemented in Vitis HLS will be responsible for performing the actual characterization of the memory, while the host code written in Python will provide a user interface for configuring and triggering the characterization process, as well as displaying the results.

# 1 Introduction

The PYNQ-Z2 is a powerful platform for developing embedded systems that combines a Xilinx Zynq System-on-Chip (SoC) with a rich set of peripherals and interfaces. It is widely used in a variety of applications, such as image and video processing, machine learning, and data analytics.

This report will describe the design and implementation of the main memory characterizer tool for the PYNQ-Z2 platform, including the hardware and software components. We will also present experimental results demonstrating the effectiveness of the tool in characterizing the performance of DDR memory modules on the PYNQ-Z2.

In this report the following topics will be covered:

1. The design of the IP block using Vitis HLS, including the source code with his testbench, and a brief discussion on the optimizations that were made by Vitis HLS to the design during synthesis.
2. The integration of the IP block into the overall block design using Vivado.
3. The implementation of the host code in Python, including the user interface, the communication with the IP block and experimental results.
4. A final discussion of the challenges and limitations encountered during the development of the DDR memory characterizer.

The list of referenced sources will appear at the end of the report.

## 2 IP block

In this section of the report, i will discuss the design of the IP block using Vitis HLS (High-Level Synthesis) that compute the bandwidth performance of the main memory. The IP block designed for this benchmark tests is a simple function that reads and writes data to and from a memory-mapped interface.

### 2.1 HLS Source code

The function takes in three inputs: an input array called ”input”, an output array called ”output”, and an integer ”modo” that controls the mode of operation.

```
void ddrbenchmark2(TYPE* input, TYPE* output, const int modo)
```

The function uses pragma statements to specify the interfaces between the function and the memory, AXI buses and control port. These interfaces are used to map the function’s input and output arrays to memory-mapped AXI interfaces for high-speed data transfer, and an AXI Lite interface for control signals.

```
//arr_depth is defined as the size of the input array
#pragma HLS INTERFACE m_axi port = input depth=arr_depth offset =
    slave bundle = gmem
#pragma HLS INTERFACE m_axi port = output depth=arr_depth offset =
    slave bundle = gmem
#pragma HLS INTERFACE s_axilite port = modo bundle=control
#pragma HLS INTERFACE s_axilite port = input bundle=control
#pragma HLS INTERFACE s_axilite port = output bundle=control
#pragma HLS INTERFACE s_axilite port = return bundle=control
```

The function then uses a series of conditional statements based on the value of ”modo” to control its behavior. If ”modo” is equal to 0, the function reads the input array with a size set at 16000*`ap_uint<DATA_WIDTH>` where `DATA_WIDTH` is the maximum width of the AXI master interface of the board (64 bits for PynqZ2), stores the values in a temporary array called ”temp”, and then writes the values back to the output array. If ”modo” is equal to 1, the function only read the input array and stores the values in another array ”read”. If ”modo” is equal to anything else, the function writes a fixed value ”outData” to the output array. This implementation allows in the host code to choose which type of test to execute according to the needs: Read&Write

(0), Read-only (1) and Write only (2). The `pragma HLS PIPELINE` directive is used to enable pipeline optimization of the for loops so Vitis HLS will attempt to divide the loop's operations into smaller stages and execute them concurrently.

```
//using TYPE = ap_uint<DATA_WIDTH>;

TYPE temp[ARRAY_SIZE];
TYPE write;
TYPE outData=123;

if(modo == 0){
   ReadWrite:for(int i=0; i < ARRAY_SIZE; i++){ //read&write
#pragma HLS PIPELINE
               temp[i] = *(input+i);
               *(output+i) =temp[i];
            }
}else if(modo == 1){
   read:for(int i=0; i < ARRAY_SIZE; i++){
#pragma HLS PIPELINE
            write = *(input+i);//solo lettura
      }
}else{
   write:for(int i=0; i < ARRAY_SIZE; i++){
#pragma HLS PIPELINE
              *(output+i) = outData; //solo scrittura
      }
}
return;
}
```

## 2.2 Testbench

The testbench allocates 2 arrays, the first one for input (filled with random data) and the second one for output fillled with zeros, after that it calls the functions `execMultiTest` which starts executing a sequence of tests, each test executes in all 3 test modes.

```
TYPE ref[ARRAY_SIZE];
TYPE out[ARRAY_SIZE];
int seed = 3456;
std::default_random_engine rng(seed);
```

```
std::uniform_int_distribution<int> rng_dist(0, RANGE_UPPER_BOUND);
for(int i=0;i<ARRAY_SIZE;i++){
   ref[i]=static_cast<TYPE>(rng_dist(rng));
   out[i]=0;
}

...

#ifdef AVERAGE_REPS
     execMultiTest(ref,out);
#endif
```

`execMultiTest(ref,out)` calls the HW function passing the arrays and the execution mode as parameters, measuring the time it takes to execute, verifies that the output is correct calling `testCheck0`, and finally prints the timing results on the screen. at the end of all the tests, it also prints an average of the execution time.

```
   for (int i = 0; i < REPETITIONS;i++){
      auto start = std::chrono::high_resolution_clock::now();
      ddrbenchmark2(ref, out,0); //HW function
      auto end = std::chrono::high_resolution_clock::now();
      double durationb =
          std::chrono::duration_cast<std::chrono::microseconds>(end
          - start).count();
      testCheck0(ref,out, &resultb);
      tot_duration += durationb;
      std::cout << "RW result: " << i << "SUCCESS="<< resultb <<
          std::endl;
      std::cout << std::endl;
      std::cout << "RW time:" << durationb << std::endl;
      rep++;
      resultb = true;
   }
   std::cout << "simultaneous RW average:" << tot_duration/rep <<
       std::endl;
```

For simplicity only the execution of the Read&Write tests is reported.

## 2.3 Vitis HLS Synthesis

The output message from the Vitis High-Level Synthesis (HLS) tool related to the implementation of the `ddrbenchmark2_Pipeline_ReadWrite` module.

The message indicates that the HLS tool started scheduling operations of the loop 'ReadWrite', in order to implement it in an optimized way. The pipelining result shows that the loop was pipelined with a final initiation interval of 1 and a depth of 3, which means that one iteration of the loop is executed per clock cycle and that there can be up to 3 iterations in progress at the same time

```
INFO: [HLS 200-42] -- Implementing module
    'ddrbenchmark2_Pipeline_ReadWrite'
INFO: [HLS 200-10] ----------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'ReadWrite'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA, Final II
    = 1, Depth = 3, loop 'ReadWrite'
INFO: [SCHED 204-11] Finished scheduling.
```

Similarly to before the final result for the loop "write" is that the loop was pipelined with a final initiation interval of 1 and a depth of 2, allowing up to 2 iterations of the loop to be in progress at the same time.

```
INFO: [HLS 200-42] -- Implementing module
    'ddrbenchmark2_Pipeline_write'
INFO: [HLS 200-10] ----------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'write'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA, Final II
    = 1, Depth = 2, loop 'write'
INFO: [SCHED 204-11] Finished scheduling.
```

## 2.4 Block design and bitstream generation on Vivado

After exporting the RTL (register-transfer level) representation of the design from Vitis HLS i moved on Xilinx Vivado to Synthesize the RTL for PynqZ2 using Vivado. This will convert the RTL into a gate-level representation (XSA in this case) of the design that can be implemented on the target device. After creating a project on Vivado i used the automation tools that Perform place and route on the imported design. This will map the design onto the target and create a bitstream file that can be loaded onto the device.

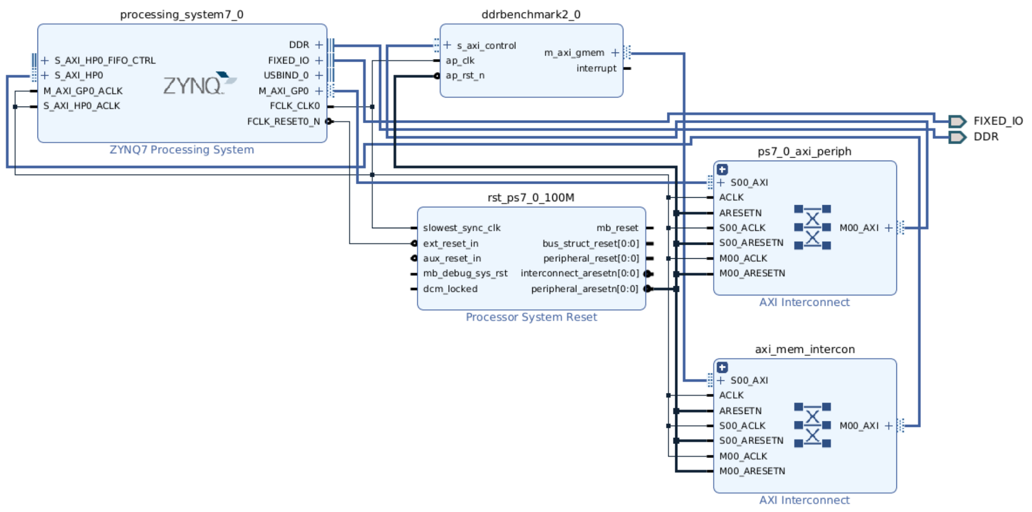


Figure 1: Final block design for PynqZ2.

## 3 System Host Code

The host code is responsible for configuring and controlling the IP block, as well as interacting with the external environment. This includes initialization, configuration of the IP block, interaction with external devices, such as memory and other IP, and collecting performance data. The host code for PynqZ2 is implemented in Python which allows for easy integration with the PYNQ framework and the use of pre-existing libraries. and executed on a Xilinx Zynq SoC (System on Chip).

### 3.1 Host code implementation in python

The code uses the Overlay class from the pynq module to load the overlay, which is a bitstream file containing the design of the IP block implemented in Vitis HLS. The code defines several variables, such as `TEST_NUM`, `DATA_SIZE`, `MB_CONV`, `MICRO_CONV` which are used as input size, test number and constants for the further performance evaluation. It then uses the allocate function from the pynq module to create two numpy arrays `buf_in` and `buf_out`, these array represents the input and output buffer for the IP block. The first argument passed to the allocate function is the size of the buffer in bytes, which is given by the `DATA_SIZE` variable, and the second argument is the data type of the buffer, which is set to `np.int64` for both arrays. These allocated buffers can be used to store the data passed to the IP block, and the results generated by the IP block can be stored in the output buffer. The final variable created is `ddrbench_ip`, this variable will be used to interact with the IP block and configure the IP block through AXI-Lite interface. Finally it calls the function called `multipleTests` that performs multiple tests of a read-write,read-only,write-only operations.

```python
TEST_NUM = 100
DATA_SIZE = 16000
MB_CONV = 1000000
MICRO_CONV = 1000000

#bitstream upload
overlay = Overlay("./ddrbench_golden_wrapper.xsa")

ddrbench_ip = overlay.ddrbenchmark2_0

buf_in = allocate(DATA_SIZE,np.int64)
buf_out = allocate(DATA_SIZE,np.int64)
multipleTests(buf_in,buf_out,TEST_NUM)
```

The function starts a for loop, that runs for the number of times specified by the repetition input argument. The for loop starts by generating random integers for the input buffer using numpy's `random.randint()` function, and assigns the values to the input buffer. The output buffer is also resetted.

```python
def multipleTests(buffer_in,buffer_out,repetition):
    sum =0.0
    rep=0

    for i in range(repetition):
        input=np.random.randint(low=0,high=8192,size=(DATA_SIZE),dtype=np.int64)

        buffer_in[:]=input.ravel()[:]
        buffer_out[:]=0
```

Then the physical address of the input buffer is written to the AXI-Lite address 0x10 and the physical address of the output buffer is written to the AXI-Lite address 0x1c. The function sets the value 0 (for this portion of code, infact, the function performs n=repetitions tests for each mode) on the address 0x28, this is used as input for the IP modo selection. It then starts a timer by calling the time.time() function, and sets the value 1 on the address 0x00 of the IP to start its execution. The function waits for the IP to finish its execution by checking if the value on address 0x00 bit 2 is equals 1, this indicates that the IP has finished. Once the IP has finished the execution, the timer is stopped and the output buffer is invalidated.

```
        ddrbench_ip.write(0x10,buffer_in.physical_address)
        ddrbench_ip.write(0x1c,buffer_out.physical_address)
        ddrbench_ip.write(0x28,0)
        start = time.time();
        ddrbench_ip.write(0x00,1)
        while(ddrbench_ip.read(0x00) & 0x04 !=0x04):pass
        end = time.time()
        buf_out.invalidate()
```

The function then checks if the input buffer is equal to the output buffer, this step is done to check if the output is correct, if it is correct the performance data is collected and printed. After all the repetition the function prints out the average time and Bandwidth of all the correct test, also the best and worst performance.

```
        if(np.all(input == buffer_out)):
            if(i==0 or
                (end-start)*MICRO_CONV<best):best=(end-start)*MICRO_CONV;
            if(i==0 or
                (end-start)*MICRO_CONV>worst):worst=(end-start)*MICRO_CONV;
            print("test ",i,":
                ","time:",(end-start)*MICRO_CONV,"micro s","
                Bandwidth:",(DATA_SIZE*64/8)/((end-start)*MB_CONV),"MB/s")
            sum = sum + end-start
            rep = rep+1
    print(" ")
    print("Executed",rep,"tests","Average_time:",
        sum/rep*MICRO_CONV,"micro s")
    print("Best_time:", best,"micro s"," Worst_time:",worst,"micro
        s" ," Avg_Bandwidth:
        ",(DATA_SIZE*64/8)/((sum/rep)*MB_CONV),"MB/s")
    print(" ")
```

For simplicity only the execution of the Read&Write mode tests is reported, the other executions mode are very similar.

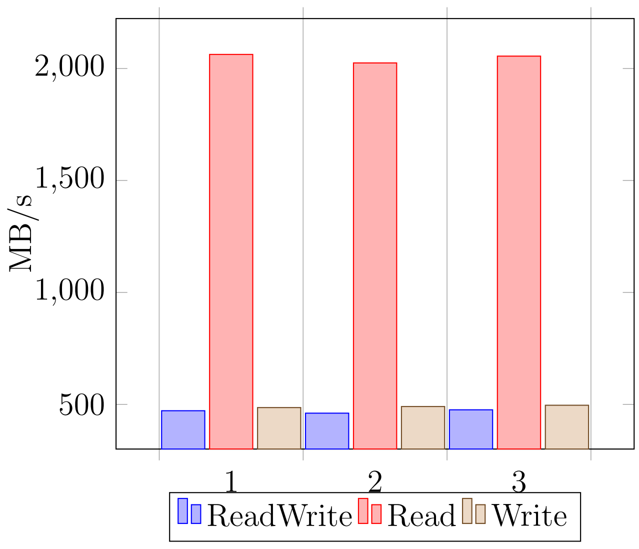
## 3.2 Experimental results

The `multipleTests` function was used to evaluate the performance of the IP block in 3 different modes of execution: mode 0, mode 1 and mode 2. The tests were executed 100 times for each mode and the average memory bandwidth was calculated.

1. In mode 0, the IP block performs both read and write operations on the input and output buffers. The average memory bandwidth for this mode was 471MB/s.
2. In mode 1, the IP block performs a read-only operation on the input buffer. The average memory bandwidth for this mode was 2011MB/s.
3. In mode 2, the IP block performs a write-only operation on the output buffer. The average memory bandwidth for this mode was 485MB/s.

For this conclusion we can see that the IP block achieved the highest memory bandwidth in mode 1 which is read-only operation. This is probably because the read operation does not have to wait for data to be written to the output buffer, as it does in mode 0. The write-only operation of mode 2 shows a good performance, although it is not as good as mode 1, it is still good and can be useful depending on the application.

In The histogram the function had been called three times.


MB/s
ReadWrite
Read
Write