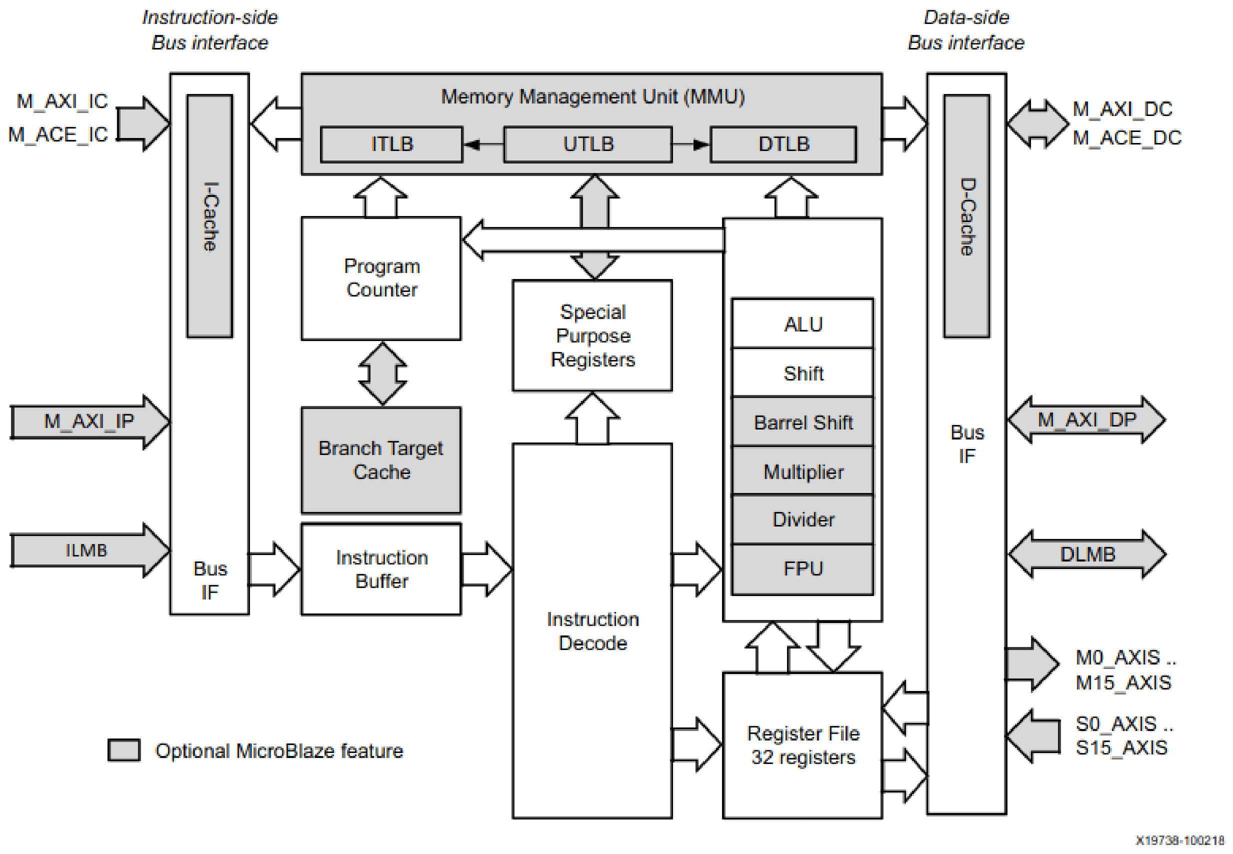
FPGA Lab – Polled and Interrupt IO on Xilinx MicroBlaze

26.06.2024

Background

Microblaze [1] is a highly customisable Intellectual property (IP) core for a RISC microprocessor developed by Xilinx. It is designed for embedded applications, thus some architectural features are best understood through this perspective. Here is a succinct list of the key points:

- Single issue in-order pipelined processors with configurable number of stages (3/5/8)
- Optional integer and floating point unit
- Local memory bus with single cycle latency and optional AXI4 instruction and data caches
- Optional AXI4 (lite) memory-mapped IO Bus
- Optional memory management unit, capable of hosting operating system



Objectives

The FGPA laboratory session is divided into two parts. The first part is described in this document and aims to familiarise you with MicroBlaze by studying basic IO concepts. In brief:

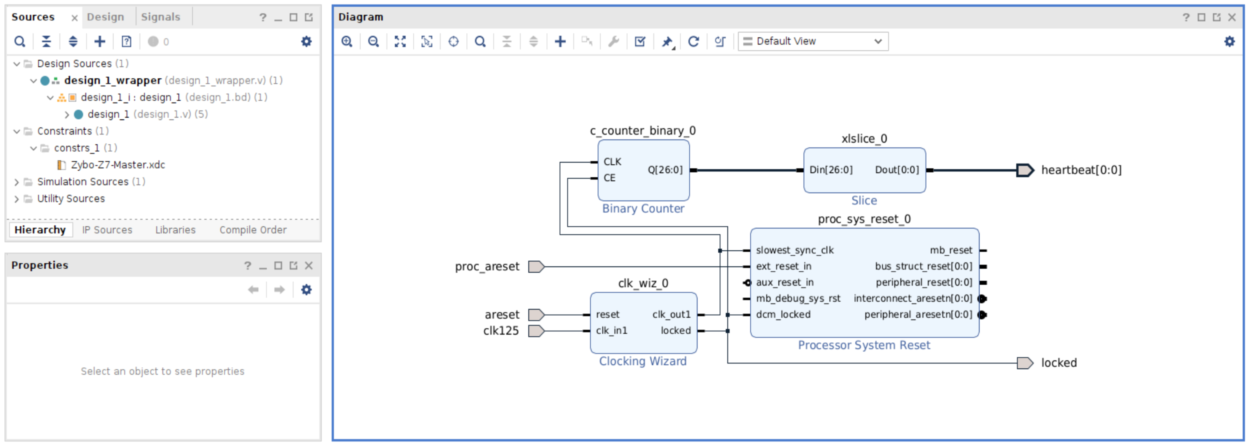
1. Set up a Vivado project targeting the Zynq 7000 All Programmable System On Chip (APSoC) device and instantiate a minimal MicroBlaze configuration.
2. Set up the software development environment using Vitis and write firmware in C to operate the processor.
3. Study the architecture and consequences of polled and interrupt-driven IO using a UART.

The second part includes an external DDR3 memory, a data cache, and investigates the memory hierarchy.

Getting Started

Clone or update the usual GitHub repository: https://www.github.com/lukevassallo/ACA24.git Follow the next steps to set up a minimal Vivado project.

1. Launch Vivado
2. Using the TCL console in the welcome screen change directory (cd) xlnx_mb.
3. Source build_bd.tcl located in the scripts directory with source ./scripts/build_bd.tcl



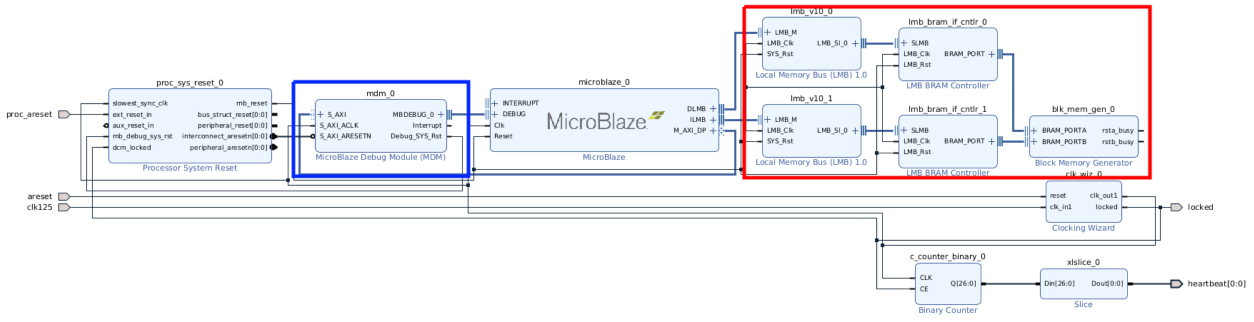
The script will set up a project for the target part (XC7Z010-1CLG400C). Afterwards, it creates a block diagram and instantiates key IP blocks for clocking and reset. IO ports are created (consistent with the constraints in the .xdc file) to bring the clock and reset signals from the board into the FPGA.

A counter is additionally created, and tied to a board LED to visually communicate the presence of the clock signal. The reference manual and board schematics contain relevant information for these steps.

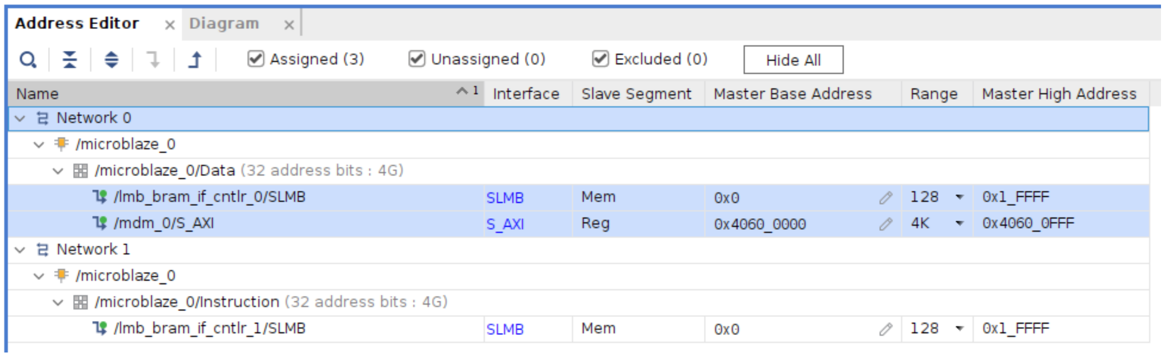
Adding MicroBlaze

Here we instantiate and connect the MicroBlaze processor, memory and debug module. The debug module contains a debug controller which provides functionality for programming and debugging the processor.

1. From the IP Integrator, add the MicroBlaze processor to the block diagram (Do not use automatic connections).
2. Add a shared program and data memory and connect it to the processor using the Local Memory Bus (LMB). The step requires three blocks:
    1. Using the block RAM generator, generate a true dual port memory. The size is not important as it will be configured later.
    2. Instantiate a block RAM memory controller to 'speak' to the memory
    3. Also, add a bridge between the processor's local memory bus and the memory controller.
3. Add debug module
    1. The debug module provides a debug controller and an optional UART. Configure the debug IP by enabling the UART, and afterwards notice the AXI slave port on the module.
    2. Use the advanced tab in the MicroBlaze IP to enable the AXI data port.
    3. Connect the processor's data port (IO Bus) to the debug UART.



4. After wiring up all the miscellaneous connections, navigate to the address tab and assign memory locations for the local memory and UART (A snippet of the memory map is illustrated below).
    1. The base address of the local memory should correspond with the processor's reset vector, which by default is 0x0000_0000.
    2. Set the local memory size for both instruction and data to 128kB. This will automatically size the block memories.
5. Wire any remaining connections. The final block diagram representative of the SoC is illustrated above. Validate the design, and generate the bitstream.



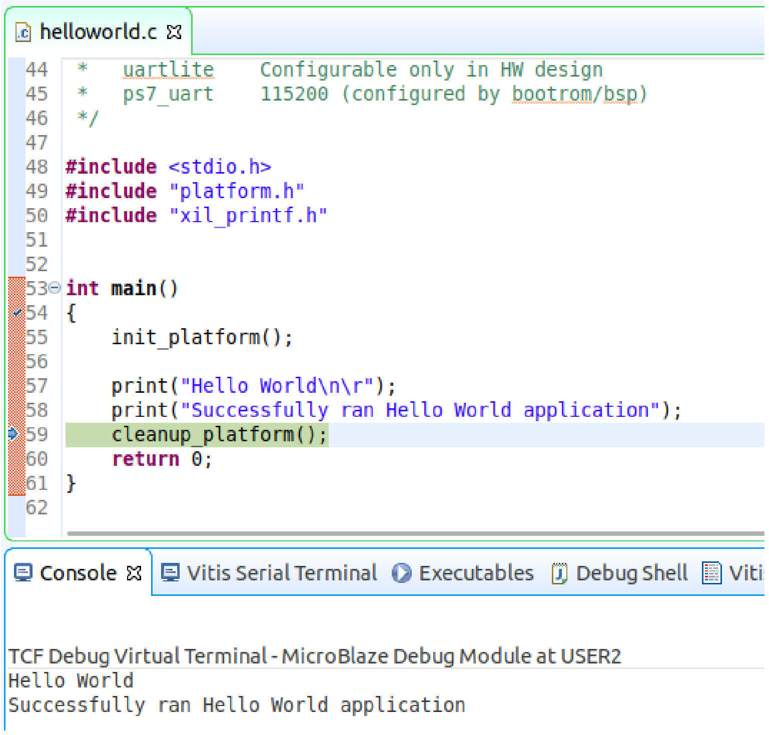
Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/microblaze_0					
/microblaze_0/Data (32 address bits : 4G)					
/lmb_bram_if_cntlr_0/SLMB	SLMB	Mem	0x0	128	0x1_FFFF
/mdm_0/S_AXI	S_AXI	Reg	0x4060_0000	4K	0x4060_0FFF
Network 1					
/microblaze_0					
/microblaze_0/Instruction (32 address bits : 4G)					
/lmb_bram_if_cntlr_1/SLMB	SLMB	Mem	0x0	128	0x1_FFFF

Writing Software

We have created a hardware platform, and the next step is to configure the environment and write software for it. This step of the process is done using Vitis.

The software environment requires knowledge of the underlying hardware to be able to compile code and program the platform. The hardware platform performs this from the Vivado-generated .xsa file. Thus creating an application requires two steps, first creating the hardware platform, and afterwards creating the application project where the C code is provided.

As a sanity check for ensuring that your hardware is working correctly create an application using the hello_world template, compile it and launch a debug session to monitor it. If successful the following should appear in the console.



```
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#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;
}

```

```
TCF Debug Virtual Terminal - MicroBlaze Debug Module at USER2
Hello World
Successfully ran Hello World application
```

Next, create an empty project and afterwards copy the main.c file for polled_io to the src directory of the application project. In the following steps, we will be issuing transactions to the memory to configure and print text using the UART.

The UART inside the debug module has only four memory-mapped registers for configuration, status and operation. Use the product guide to configure these registers and print the text string to the console. The document can be easily obtained by right-clicking the IP block in IPI, or on the top left corner of the IP Configuration window. It is also available in the Document Navigator, installed together with the tools.

Improve the application by echoing the characters received by the UART. Monitor the status of the UART, and print characters that are received by the UART.

Homework Part 1/2 – Interrupt IO

Go back to Vivado, and connect the interrupt line of the debug module to the processor. Generate the bitstream and update the hardware platform in Vitis with the new .xsa file.

Create a new application project, based on the provided interrupt_io code. Use interrupts such that an infinite while loop is relieved from IO operations and can be used to perform useful work. Here are some hints to help you get going:

- When the processor receives an interrupt, it will pause its current operations, back up the register file and branch to the interrupt vector. We can associate a function with the interrupt vector that is automatically invoked to perform critical code operations.
- The processor does not react to interrupts by default, and they will need to be enabled explicitly. The hardware platform contains MicroBlaze drivers for configuring interrupts amongst many other features. Include “xil_interrupts.h” and then use the appropriate function to enable interrupt processing.

This part of the homework contributes 40% of the final mark. Full guidelines are provided in the second document.

References

1. MicroBlaze Processor Reference Guide (UG984)