Microprocessor-based system simulation and debugging function implementation

Bovolenta, Casciano, Nino

March, 2021

1 Introduction

Primary objective of this project is to create a fully working system, starting from the architecture of a microprocessor Cortex-M3 and simulating the environment around it. To reach our goal we had to simulate, with a testbench, the behaviour of a memory and then link it to the microprocessor through the AXI interface.

The final part of our effort has been put in the implementation of a debugger. We implemented a debugger that allows us to visualise and analyse more clearly which are the specific signals exchanged between processor and memory, during the execution of a specific portion of code. In this way we could have a low level understanding of the mechanisms behind the execution of the code from the hardware point of view.

The following paragraphs aim to describe how we implemented the testbench, the problems we encountered and the decisions taken.

2 Testbench and architecture

2.1 Memory address space

The core of our project is a testbench implemented in Vivado. It includes a real memory address space, an AXI4Stream interconnect, and a system to address data into the correct portion of the memory. The role of the testbench is to emulate the environment around the Cortex-M3 processor in order to guarantee its correct operation.

The first thing we did was the implementation of the memory address space. To do this, we took the classical structure of a single address space for a 32-bit microcontroller architecture and implemented each section of the memory as a separated matrix with the smallest possible size, in order to produce a lighter simulation.

As a reference, we took the memory layout of STM32F107RB microcontroller. Its structure is the following:

```
logic [7:0] boot          [18'h00000 : 18'h3FFFF]; // if MSB = 00
logic [7:0] flash         [18'h00000 : 18'h3FFFF]; // if MSB = 08
logic [7:0] sys_mem       [15'hB000  : 15'hFFFF];  // if MSB = 1F
logic [7:0] ram           [17'h00000 : 17'h10002]; // if MSB = 20
logic [7:0] peripherals   [18'h00000 : 18'h29FFF]; // if MSB = 40
logic [7:0] m3_registers  [20'h00000 : 20'hFFFFF]; // if MSB = E0
```

As listed above, we used the most significant byte to select the correct memory section, thanks to a set of functions implemented in the testbench. For what concern the least significant bytes, they identify the specific address in the previously selected memory section.

2.2 Memory description

Each portion of the address space represent a different function:

- boot: part of the address space that is aliased, during the startup of the system, to flash or system memory.
- flash: non volatile memory that contains the executable binary file of our software.
- system memory: memory where the boot loader is located. It is used to reprogram the flash memory with a serial interface.
- ram: volatile memory used during the execution of the program.
- peripherals: peripherals are memory mapped hardware components. This is the portion of the address space where the processor has to write or read in order to communicate with them.
- m3_registers: internal peripherals of the processor are mapped here.

2.3 Memory initialization

The program counter of the processor starts from the address 0x00000000 and, in a complete architecture, this part of the memory address space is aliased to the flash memory or to the system memory depending on boot pins. Since we didn't have the necessity to really use the system memory to reprogram the flash, we decided to introduce a fixed alias to the flash.

In order to do this operation, when the processor asks to read from an address of the boot memory (when the MSB is 0x00), we consider only the LSBs of the address requested and we provide the data contained in the corresponding address of the flash memory.

2.4 Bus architecture

The processor contains 4 bus interfaces:

- ICode memory interface: instruction fetches from Code memory space (0x0000000 - 0x1FFFFFFF) are performed over this 32-bit *Advanced High-performance Bus* (AHB)-Lite bus.
- DCode memory interface: data accesses to Code memory space (0x00000000 - 0x1FFFFFFF) are performed over this 32-bit AHB-Lite bus.
- System interface: instruction fetches, data and debug accesses to System space (0x20000000 - 0xDFFFFFFF, 0xE0100000 - 0xFFFFFFFF) are performed over this 32-bit AHB-Lite bus.
- External *Private Peripheral Bus* (PPB): data and debug accesses to External PPB space (0xE0040000 - 0xE00FFFFF) are performed over this 32-bit *Advanced Peripheral Bus* (APB) (AMBA v3.0) bus.

We connected to our testbench the two bus interfaces provided by the Cortex-M3 IP-Core: CM3_CODE_AXI3 (used for instruction fetching and data accessing in addresses 0x00000000 - 0x1FFFFFFF) and CM3_SYS_AXI3 (used to access to system space in addresses 0x20000000 - 0xFFFFFFFF).

2.5 GPIO function

After understanding the architecture and how to manage data exchange between the processor and the memory, we implemented a simple mechanism to emulate the operation of the GPIO, without emulating or implementing the complete peripheral.

The main idea was to introduce some instructions in the C code that allowed us to write "1" or "0" in a specific address of the memory so that, in our testbench, we could observe the value stored in that specific address. In this way we could add, in the behavioral simulation, the signal corresponding to the value stored in that address and use it to visualise the debugging function.

C code example:

```
volatile uint32_t* __DEBUG = (uint32_t*)0x40010800;
//CTR_GPIOA address, only as reference

*__DEBUG = 1;
//block of code
*__DEBUG = 0;
```

SystemVerilog code example:

```
logic debug_led;

always@(*)
begin
    if(rst == 1'b1)
```

```
6       begin
7         debug_led = peripherals[18'h10800];
8       end
9   end
```

3 Debug

In the following we will analyse more in detail the debug, whose operation was introduced at the end of the previous section.

3.1 General structure

A debugger is a computer program that allows to step through a program, line by line. By analysing the program flow, the programmer can control the behaviour of his code and find errors that are visible only during run-time operation.

In general, a debugging tool offers five main functionalities:

- Stepping: the programmer can observe the behaviour of the code, at each line;
- Variables and expression inspection;
- Breakpoints: the programmer can interrupt the execution of the code in some specific points;
- Watchpoint: the code can stop, if a particular variable has changed its value;
- Memory view: the programmer can have an idea of the memory utilization and where variables are stored.

3.2 Custom debugger description

Our debugger presents only the last functionality, with some in depth observation of the signals and information exchanged between processor and memory.

The idea is to "delimit" a particular piece of code or some pieces of code, through a "debugging" variable. As previously explained, this variable is nothing but a pointer, which points to an address that the programmer can choose. Its content is modified before and after the desired section, according to a specific pattern that the programmer decides. For instance, to highlight the execution of a "for" loop, the debugging variable assumes the value "1" before the loop, and it is reset to zero after the loop; this behaviour can be adopted every time a check of a "for" loop is needed. Instead, if the execution of an "if" statement must be performed, the variable could be set and reset twice, before and after the statement.

We can easily understand that our debugger behaves like a silent observer, because it does not alter the normal flow of the program. Since our debugging

tool is not a real program, we used the simulation of the testbench as a "display" to see the execution of the code. Depending on the address we had chosen for the debugging variable, we had to inspect the AW bus (which stands for "Address Write" bus) and the W bus (which stands for "Write" bus), belonging to either "CODE" or "SYSTEM" section of the buses of the microprocessor.

On these two buses, we could see the content of the debugging variable and, consequently, we were able to recognize which part of the code the microprocessor was executing. To have a much faster visualization of the changes involving the debugging variable, inside the testbench we introduced a signal, which varies accordingly with the variable. The programmer can also decide to use more than one debugging variable and to save in it, for example, the value of another variable; this is what we did to test our debugging tool and to understand if our code was running correctly. We defined a variable named "k" and, in our code, its value was incremented by one, every iteration of a "while" loop. In order to be sure that this operation was really performed, we decided to use two debugging variables: one to save the value of "k" and another to save the address in which it was stored. We would have expected to see that the microprocessor would ask to read the address of "k", first, and then it would write the new value of "k", instead we could not observe any read and write operation involving the address where it was stored. Digging deeper into the microprocessor functioning, we found that Cortex-M3 uses the so called "bit-banding" technique to modify the content of a register and, consequently, of a memory address.

3.3 Bit-banding

Usually to change the content of an address, the microprocessor performs a "read-modify-write" operation; this means that first it reads and saves, in a temporary register, what is contained inside the address, then it updates it and finally it writes the new value in memory. It is easy to understand that this approach is not so good for two reasons: it requires a lot of memory and if an interrupt occurs in between two operations, what will be stored in memory could be wrong. The problem is that it is not performing an atomic operation; bit-banding tries to overcome this issue.

SRAM memory is divided into two regions: 1MB region (starting from address 0x20000000) called "bit-band region" and 32MB region called "alias region" (placed after "bit-band region"). Each address in the "alias region" is mapped to a single bit contained in an address of the "bit-band region". If we need to modify a bit inside an address in the "bit-band region", we have to write into the corresponding address of the "alias region". Thus, the update of the address content becomes an atomic operation. This is the reason why we could not see the address of "k" in the read/write bus.

There is a formula that, starting from the address in the "alias region", allows to understand which bit of which address in the "bit-band region", we are modifying. The programmer can implement an algorithm to automatize the "decryption" process; this can be an idea to continue and improve this project.

4 Conclusion

Our starting point was the microprocessor Cortex-M3, available in the form of IP core. By exploiting a testbench, simulating a memory and an AXI interface, we were able to make it working as a real microprocessor. Finally, to be sure that the Cortex-M3 was executing correctly our application, we implemented a very simple and rough debugger. Its purpose was to investigate, at low level, the flow of information and the signals exchanged between memory and microprocessor. Since it is well known that there are many debugging tools and, in general, they use the UART interface, at the end of our report, we want to explain why we decided to use such a debugger and what are the main advantages.

First of all, none of the debuggers available nowadays allows the programmer to analyse, in such a simple way, the signals exchanged between microprocessor and memory; in most common cases, the programmer can inspect the information that flow in and out of the microcontroller, but he cannot see the interaction of its internal microprocessor with the memory. Moreover, an external physical device is needed to connect the microcontroller to the PC and to debug it. Since our debugger is a tool used in a hardware simulation, it does not need any additional device, which could not be available. Another relevant aspect to be mentioned is that, with this type of debugger, the programmer is free to create his own protocol; this means that he can define a particular behaviour of the debugger, for each section of the code he wants to analyse. As a final remark, it is important to underline that this debugger can be used even in cases where area occupation is a concern, because it does not require the implementation of any internal debugging logic, like the ILA (Integrated Logic Analyzer) in Vivado.

To conclude, we can state that the project can be further improved. A possible idea could be the implementation of an algorithm used to manage the bit-band feature, in order to translate the "alias region" addresses, in "bit-band region" addresses. This operation can be useful in order to have a better understanding of the signals we are observing and to organise and handle them with a custom software.

Another way to continue the project could be the emulation of some peripherals, such as GPIO or UART, in order to create a complete microcontroller system.