Embedded Systems - project report

Alessia Guzzo
Chiara Muscari Tomajoli
Matteo Cattaneo

March 2021

Contents

1 Introduction

We thought it would be interesting to analyse the different memories from which the Cortex_M3 can read instructions. We looked for information on built-in memory, the so called ITCM, in the Cortex design-start guide. Its dimension can be at most 1 MB and for initializing it is necessary to load an initialization file (.hex). In particular, the processor will fetch instructions starting from address 0x00000000, whatever will be mapped to that address. We discovered that it is actually possible to have the instructions to be executed by the processor stored and fetched in a memory external to the processor and this can be done by setting to '0' the CFGITMEN[0] signal. Therefore, we can change the device from which the instructions are fetched by manually changing the address of our instruction memory using the address editor: the one that is located at address 0x00000000 will be the one from which instructions will be fetched.

Figure 1: signal CFGITCMEN

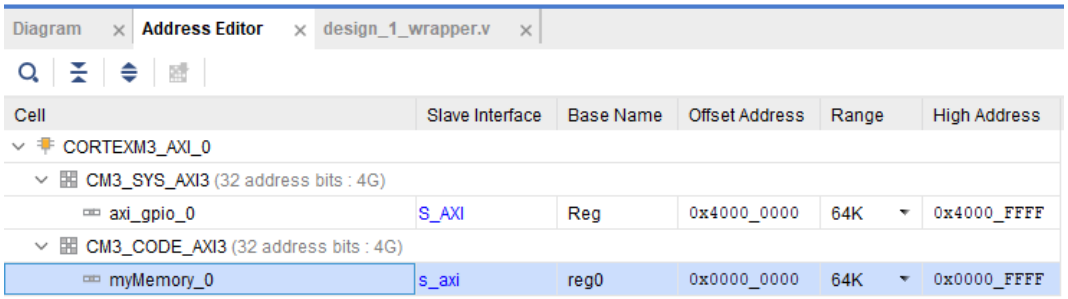

Figure 2: address of the fake memory changed to 0x00000000 manually

2 Implementing instruction memory

Exploiting the ITCM, which is implemented inside the Cortex-M3-IP-Core, we cannot look at the data exchanges between CPU and instruction memory unless using the debugging function of the Cortex-IP-Core. This is the reason because we have chosen to do all the comparisons between a BRAM and a new memory built by us.

2.1 BRAM as instruction memory

To use the BRAM as instruction memory, we had to instantiate a block memory generator module and a bram_controller, which communicates with the processor.

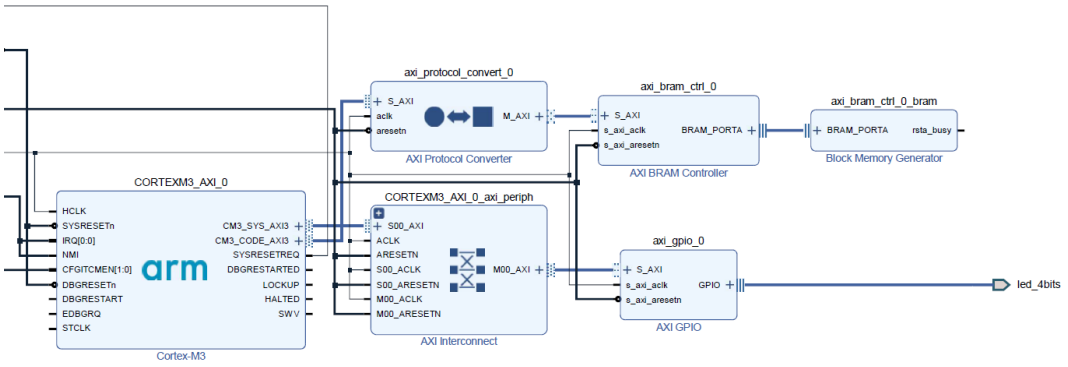

Figure 3: block design of BRAM memory as instruction memory

2.2 Fake memory as instruction memory

We built a new distributed memory module as a Finite State Machine, which sends and receive data using the AXI4-LITE interface. We tested it trying to make the same writing and readout operation that was made in DesignStart of the BRAM module.

Figure 4: read and write of AXI4-LITE transition

At the end we instantiated this memory as an instruction memory connecting it through the CM3_CODE_AXI3.

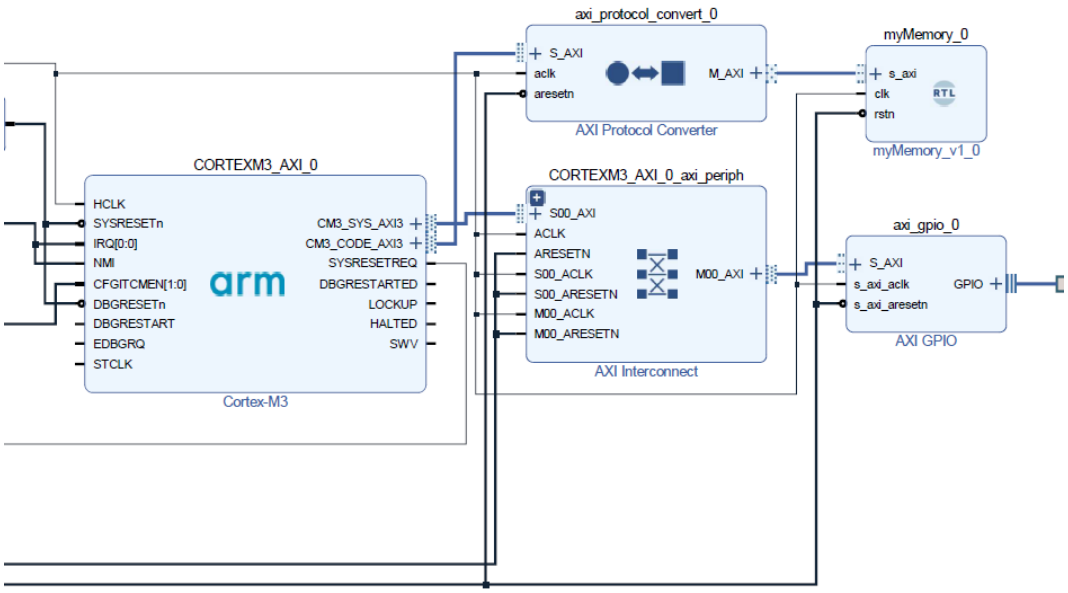

Figure 5: block design with our memory as instruction memory

3 Simulations

3.1 Timing comparison

Our objective for this session was to extract, from the post-synthesis and post-implementation simulation, as much information as possible regarding the timing of the two memories, the BRAM and the one built by us using LUTs. Thus, to estimate which memory is faster, we looked at the delay between the rising edge of the clock of the axi_protocol_converter and the change in the RDATA bus, which is the response of the memory to the address required by the processor. In the following list we report our times:

- Post-Synthesis Simulation
  - BRAM memory : 2,26ns
  - Our memory : 0,9ns
- Post-Implementation Simulation
  - BRAM memory : 2.5ns
  - Our memory : 2.16ns

3.2 Disassembler

We thought it could be interesting to provide some more precise timing information about the instructions fetched and executed by the processor and, regarding this, we found a functionality of keil called "Disassembler", which translates the .c firmware in its assembler translation, enabling us to follow with more precision the path of a single instruction. On the RDATA port of the CM3-CODE-AXI port, which is the one that fetches the instructions from the external memory, we found the instruction which stores the value "leds on" ('1111') to the memory address corresponding to the LEDs GPIO. Then, we found out that the corresponding write of the data on the CM3-SYS-AXI port, is after 100ns, which is five clock cycles. After that, as can be seen in Fig.8 after another 100ns (5 clock cycles) the value 0x0000000f is exposed on the leds output, turning them on.

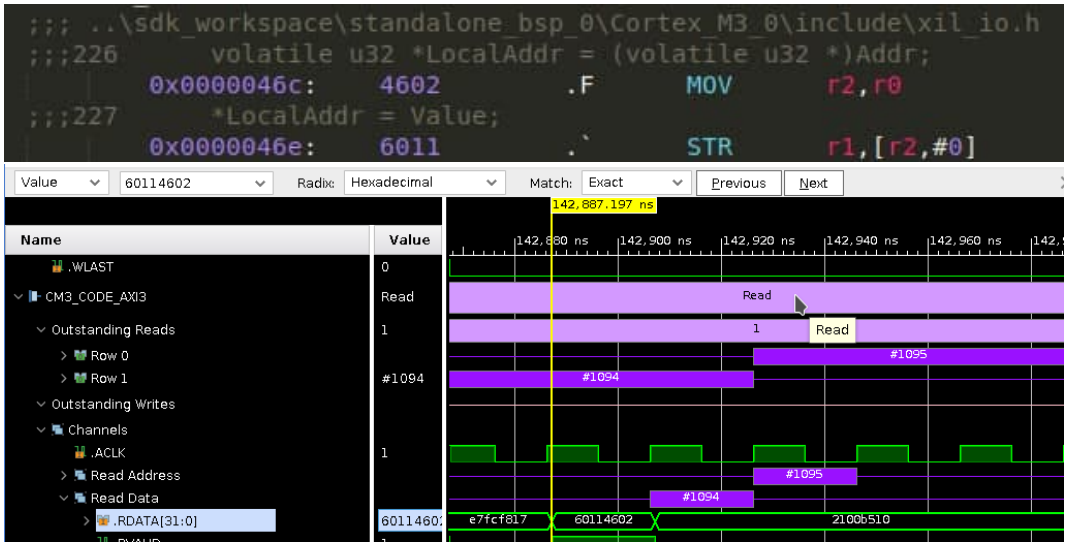

Figure 6: 60114602 corresponds to the store instruction of the leds value

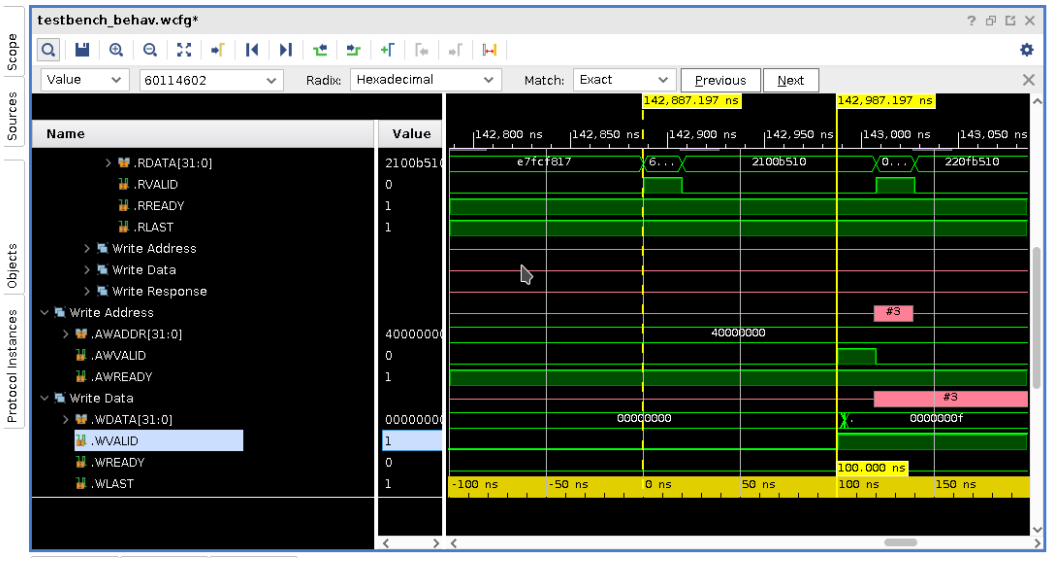

Figure 7: delay between instruction fetch and data exposed on write port

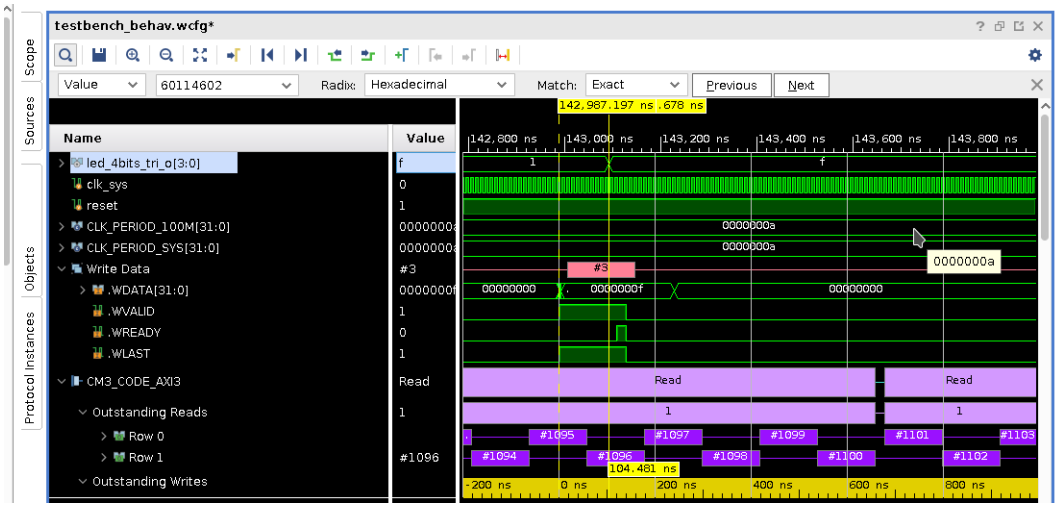

Figure 8: delay between data exposed on write port and leds turning on

3.3 Glitches

Looking at post-implementation simulation, we found out a curious behaviour of the signal controlling the LEDs. In both the high-to-low and the low-to-high transitions, we see that there are intermediate states. In particular, in the transition f $\rightarrow$ 0, where f means '1111' (all leds on) and 0 means '0000' (all leds off), leds do not turn on at the same time, but go through the states b(1011), 3(0011), 1(0001), and finally 0(0000). This behaviour is peculiar to the post-implementation simulation and is due to the different paths (which means, different delays) that the signals take. This behaviour is called Glitch and we expect this delay to change from implementation to implementation.

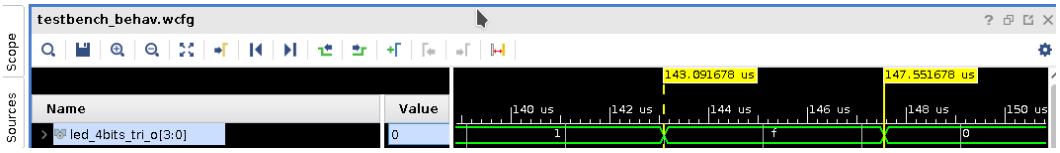

Figure 9: post synthesis

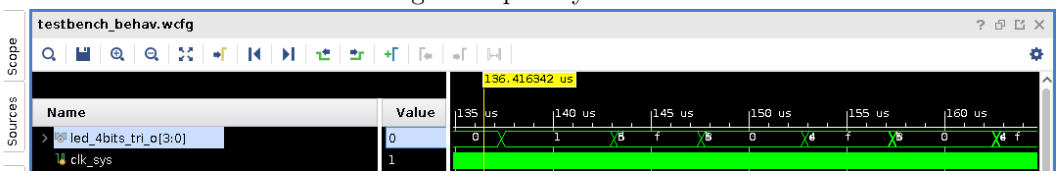

Figure 10: post implementation

In this particular implementation, we can observe that the first signal that stabilizes to 0 is the second most significant bit and, after $17ps$, the most significant bit settles to 0. Subsequently, after another $1,95ns$, the third most significant bit goes to 0 and the least significant bit goes to 0 after another $0.5ns$. The total duration of the glitch in this implementation is $2.467ns$, which should be compared with the total duration of the "leds OFF" and "leds ON" stages, that are $4,46\ \mu s$ and $5,29\ \mu s$ respectively, so they have a completely negligible duration. We asked ourselves how a design can cope with the glitch problem, in fact, even if in our case it is not a problem (it is just a led turning on sligthly after another one), it can become critical if, for example, the output signal is desired to switch at high frequency, because in this case I must be sure that my output signal is stable and that every glitch has ended. Indeed, we noticed that the WREADY signal is asserted by the slave well after the leds are stabilized on their steady state values, thus preventing the master from trying to perform any other transfer until the glitch has ended.

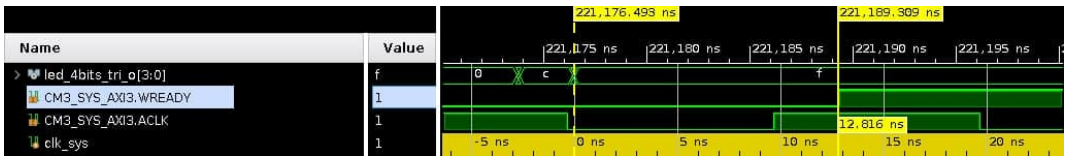

Figure 11: WREADY is asserted after the led is turned on

4 Conclusions

4.1 Timing

By comparing the timing of BRAM and our memory in Post Synthesis and Post Implementation Simulation, we can conclude that the difference of speed is not so noticeable. However, comparing post synthesis with post-implementation, a significant worsening in performance can be seen in the memory designed by us, which is made of LUTs (from $0.9ns$ to $2.16ns$). This can be explained thinking about the delays due to the routing between all the LUT module composing the distributed memory, which are taken into account only in Post Implementation simulation. As a further confirmation, we can see that the performance degradation of the BRAM is not so pronounced (from $2.26ns$ to $2.5ns$), in fact BRAM module is a (fixed) module embedded into the FPGA fabric, whose real latency is already well-modeled in post-synthesis.

4.2 Disassembler

The interesting point of the store instruction is the 100ns latency (corresponding to 5 clock cycles) that the processor shows between the fetch of the STORE instruction and the exposition of the data to be written to the LEDS GPIO on the CM3_SYS_AXI3 port. We found that this 5-cycles delay is comparable with the pipeline structure of the Cortex-M3 shown in Cortex-M3 Specification, which shows a 3-cycle pipeline. We are not able to go into more detail the internal structure of the processor, but we can assume that some stages of the pipeline, for example the execution stage, will take more than one clock cycle, resulting in more clock cycle latency than the other stages of the pipeline.