Digital Circuits Project

Matteo Arrigo

06/03/2024

1 Introduction

This project was carried out as the final assignment for the Digital Circuits course in the third year of Computer Engineering at Politecnico di Milano.

The task is to design a hardware architecture that meets the assigned functional specification, both in pre-synthesis and post-synthesis, with the only non-functional constraint being to ensure that signal propagation delays do not exceed the assigned clock period (20 ns).

The architecture interface is as follows:

```
entity project_reti_logiche is
    port (
        i_clk   : in std_logic;
        i_rst   : in std_logic;
        i_start : in std_logic;
        i_add   : in std_logic_vector(15 downto 0);
        i_k     : in std_logic_vector(9 downto 0);

        o_done  : out std_logic;

        o_mem_addr : out std_logic_vector(15 downto 0);
        i_mem_data : in std_logic_vector(7 downto 0);
        o_mem_data : out std_logic_vector(7 downto 0);
        o_mem_we   : out std_logic;
        o_mem_en   : out std_logic
    );
end project_reti_logiche;
```

The system also interfaces with a byte-addressed memory with 2^{16} words of 1 Byte each.

The operations to be performed in sequence, once started, depend on the values `ADD` and `K` provided by the inputs `i_add` and `i_k`. Memory reading must start from address `ADD` for a total of 2K words, divided into even positions (where reading starts), which indicate significant values, and odd positions, which indicate the credibility value of the previous significant value. The system's task is to replace null significant values with the last non-null significant value read, and set its credibility value accordingly. If a non-null value is read, its credibility is set to `31`, while for each consecutive null significant value read, the credibility is decremented. If credibility reaches `0`, it remains at that value without further decrementing until the first non-null significant value is read.

If a null significant value is read at the beginning of the operation sequence, and as long as it remains so, it is left null and its credibility is also set to `0`. For subsequent operation sequences and after each reset, the operating logic remains the same.

The signals `o_mem_addr`, `i_mem_data`, `o_mem_data`, `o_mem_we`, `o_mem_en` are used to interface with the memory, with the obvious meanings suggested by their names, while `i_clk` is the clock signal that synchronizes all other signals. In particular, except for the only asynchronous signal `i_rst`, all signals are to be interpreted on the rising edge of the clock. The other signals are:

1. **`i_rst`**: asynchronous signal that resets the system. `i_start` can be raised only after `i_rst` has been lowered. The system's behavior before the first time `i_rst` is set high is unspecified.

2. **i_start**: indicates the start of operations. It remains high until `o_done` becomes '1', after which it must be lowered. After both `o_done` and `i_start` are lowered, another sequence of operations can be started directly using `i_start`, without needing to reset the system again with `i_rst`.

3. **i_add**: indicates the ADD value of the functional specification.

4. **i_k**: indicates the K value of the functional specification.

5. **o_done**: signals the end of the operation sequence when it becomes '1'. Once raised, there should be no further communication with the memory, and it must be lowered after `i_start` goes to '0'.

An example of a possible sequence of such operations is given, with ADD=10 and K=10:

Addr	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Before	0	0	128	0	64	0	0	0	0	0	100	0	1	0	0	0	1	0	23	0	0	0
After	0	0	128	31	64	31	64	30	64	29	100	31	1	31	1	30	1	31	23	31	0	0

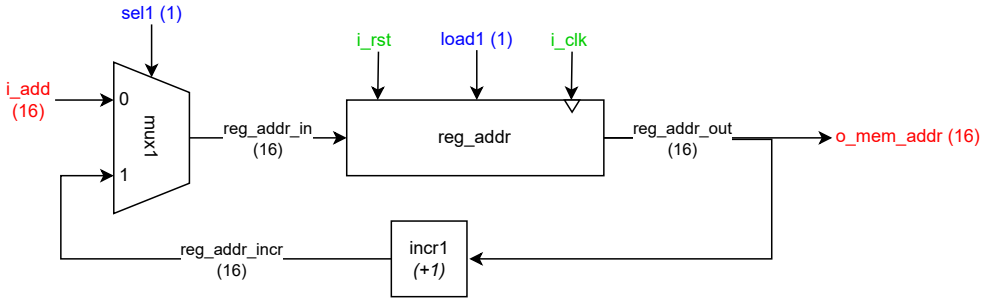
2 Architecture

Below is a description of the architecture designed and implemented for the project. The signal names refer to those used in the VHDL code of the main architecture. Overall, the architecture is divided into 3 independent modules and a state machine that manages the control signals. Input/output signals directly managed by the architecture are highlighted in red, while control signals managed by the FSM (i.e., the FSM outputs) are highlighted in blue. Green signals are the reset and clock, present in the registers, while the remaining signals are internal signals (corresponding to `signal` in VHDL). The number of bits for each signal is indicated in parentheses.

The basic architectural modules used are (with reference to the name of the respective VHDL entities):

- **mux**: a standard MUX with 2 inputs of 16 bits, 1 output of 16 bits, and a selection bit.
- **reg16**: a synchronous (flip-flop) 16-bit register, with an asynchronous reset signal and a load signal that enables writing when high.
- **addr16**: a 16-bit adder implemented using 16 full-adders (actually used only as an incrementer or decrementer).

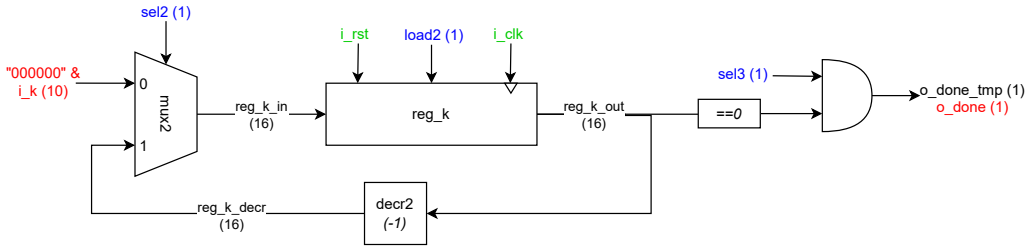
2.1 Module 1: Memory Address Manager



The bit `sel1` sends to the input of the `reg_addr` register either the signal `i_add` (for initialization) or the output of the register itself after it has been incremented (i.e., the output of the `incr1` module, which performs the +1 operation). Writing to the `reg_addr` register is controlled by the bit `load1`. The output of the register represents the `o_mem_addr` output of the entire architecture.

The VHDL implementation is fully Structural.

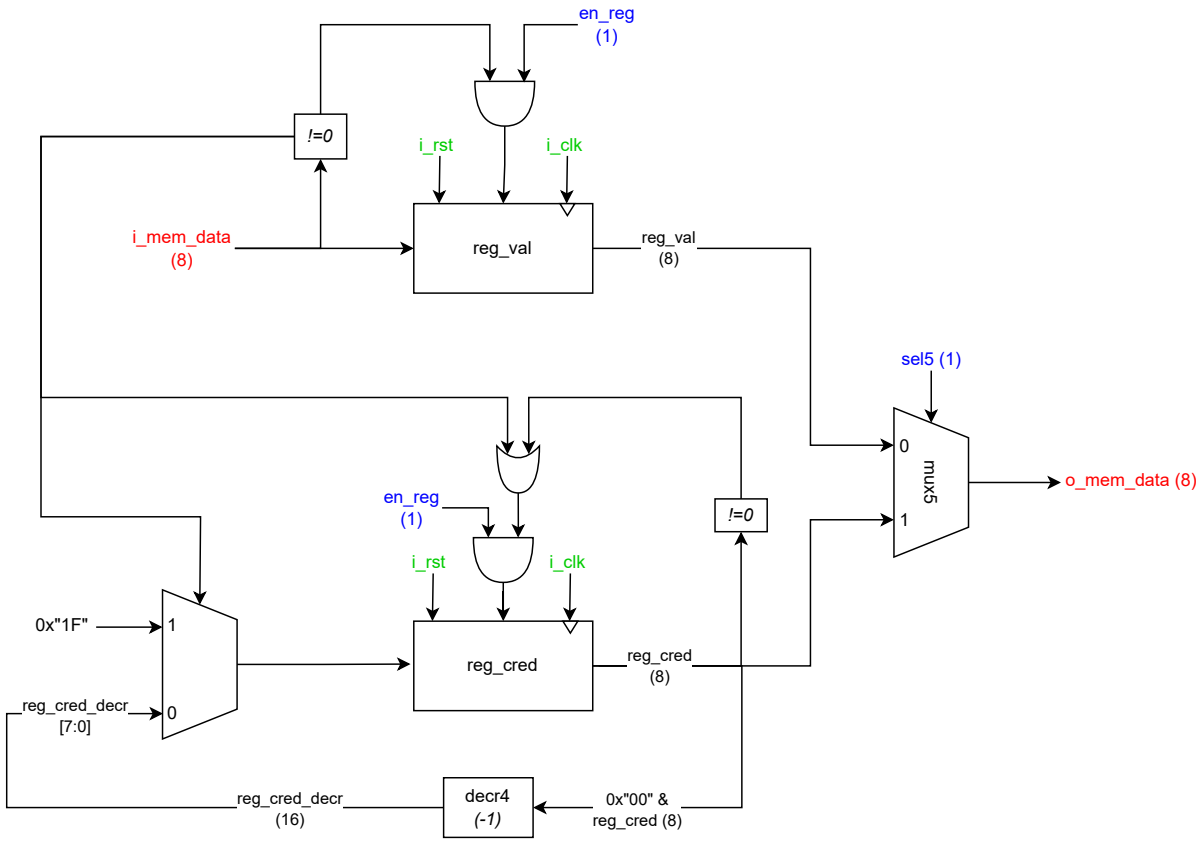
2.2 Module 2: Iteration Counter Manager



The bit `sel2` sends to the input of the `reg_k` register either the signal `i_k` (for initialization) or the output of the register itself after it has been decremented (i.e., the output of the `decr2` module, which performs the -1 operation). Writing to the `reg_k` register is controlled by the bit `load2`. The `==0` module outputs '1' if `reg_k_out` is zero, '0' otherwise, and, ANDed with the bit `sel3`, defines the output `o_done`.

The VHDL implementation is mixed, almost entirely Structural except for a Behavioural part (i.e., a `process`) to define the output `o_done` as a function of `reg_k_out` and `sel3`. The auxiliary signal `o_done_tmp` allows reading `o_done` (as an input to the FSM), which otherwise would be only an output (not readable) of the architecture.

2.3 Module 3: Memory Interaction Manager

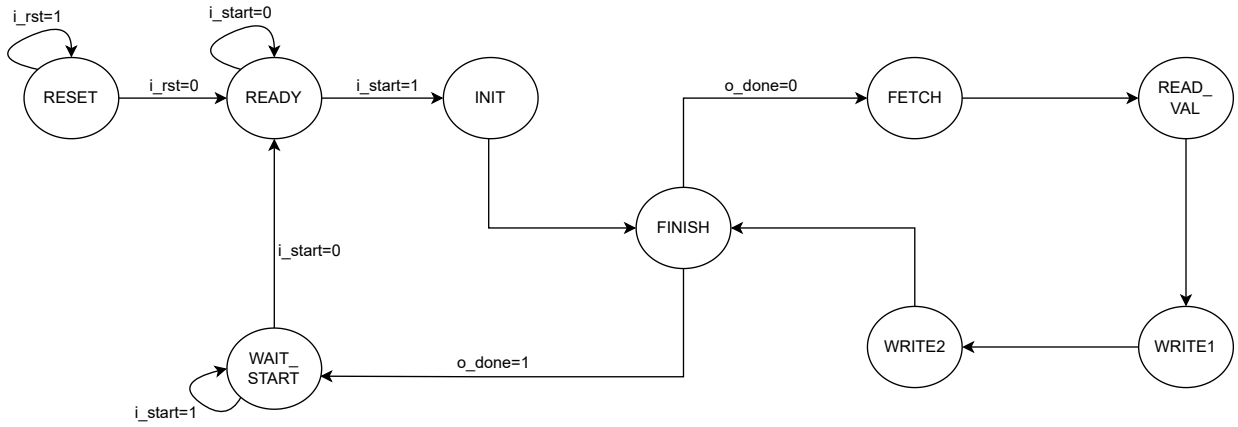


Three main parts can be identified:

- **reg_val register**: register that stores the last non-zero value read. It is enabled for writing if the bit `en_reg` is high and the value `i_mem_data` is non-zero (used to store the value if it is non-zero, otherwise the last non-zero value is kept in memory).
- **reg_cred register**: The input is either the value 31 (decimal) or its decremented output (i.e., the output of `decr4`, which performs the -1 operation). The choice of input depends on whether the signal `i_mem_data` is zero (if the value read is non-zero, we set 31 as the standard credibility value, otherwise we decrement the current credibility). Writing to the register occurs if `en_reg` is high and either `i_mem_data` or the current credibility value are non-zero (specifically, we must check `reg_cred!=x"00"` to avoid further decrementing credibility when zero is reached).
- **mux5**: selects whether to output (`o_mem_data`) the output of the `reg_val` register or `reg_cred` based on the bit `sel5`.

The VHDL implementation is almost completely Behavioural, except for the use of Structural style to manage only the `decr4` module.

2.4 FSM



	load1	sel1	load2	sel2	sel3	en_reg	sel5	o_mem_en	o_mem_we
RESET	0	1	0	1	0	0	0	0	0
READY	0	1	0	1	0	0	0	0	0
INIT	1	0	1	0	0	0	0	0	0
FETCH	0	1	0	1	0	0	0	1	0
READ_VAL	0	1	0	1	0	1	0	1	0
WRITE1	1	1	0	1	0	0	0	1	1
WRITE2	1	1	1	1	0	0	1	1	1
FINISH	0	1	0	1	1	0	0	0	0
WAIT_START	0	1	0	1	1	0	0	0	0

The state machine manages the control signals by passing through 9 states. It was designed as a Moore machine, so the outputs can be indicated in a separate table as a function of only the current state. In the table, outputs that differ from the default (as written in the VHDL FSM code) are highlighted in red.

The states are:

RESET	FSM reset state, where it remains until `i_rst` goes low again
READY	Wait until `i_start` is set high to begin the operation sequence
INIT	Set the initial values of the `reg_addr` and `reg_k` registers to start the operation sequence
FETCH	Enable memory reading to set `i_mem_data` with the significant value to be considered for this iteration
READ_VAL	Based on `i_mem_data`, set the correct values for this iteration in the `reg_val` and `reg_cred` registers
WRITE1	Write the value of `reg_val` to memory. Increment `reg_addr` to prepare for writing the credibility value
WRITE2	Write the value of `reg_cred` to memory. Increment `reg_addr` to prepare for the next iteration Decrement `reg_k` to mark the iteration as completed
FINISH	State corresponding to the end of an iteration, where there is no communication with memory. If `i_k` has not reached 0, start a new iteration from FETCH. Otherwise, move to WAIT_START
WAIT_START	Wait until `i_start` goes low again

The VHDL implementation is the standard one also seen in class, with 2 separate processes to manage the next state function and the output function.

3 Experimental Results

3.1 Synthesis

The described architecture, once implemented in VHDL, can be synthesized and simulated as if it were on a real FPGA (the one used for synthesis is the suggested Artix-7 FPGA xc7a200tfbg484-1), through post-synthesis functional simulation. In particular, we can verify certain aspects based on the post-synthesis simulation of the example testbench provided for the project.

Using the TCL command `report_utilization`, it can be observed that no latches were synthesized, as intended. Instead, 52 flip-flops are generated, which the synthesis tool likely used to manage the 16 bits of the `reg_addr` and `reg_k` registers, the 8 bits of the `reg_val` and `reg_cred` registers, and 4 bits to handle the 9 states of the FSM.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	66	0	0	134600	0.05
LUT as Logic	66	0	0	134600	0.05
LUT as Memory	0	0	0	46200	0.00
Slice Registers	52	0	0	269200	0.02
Register as Flip Flop	52	0	0	269200	0.02
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Furthermore, using the `report_timing` command, since there is a constraint that signal delays must not exceed the clock period of 20ns, it can be seen that the slack time (the time within the clock period during which signals are stable, with no operations being performed) is largely positive (16.213 ns). Therefore, the proposed architecture does not appear to have any issues regarding signal delays.

```
Timing Report

Slack (MET) :             16.213ns  (required time - arrival time)
  Source:                 reg_cred_reg[4]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            reg_cred_reg[0]/CE
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            20.000ns  (clock rise@20.000ns - clock rise@0.000ns)
  Data Path Delay:        3.405ns  (logic 1.019ns (29.927%)  route 2.386ns (70.073%))
  Logic Levels:           3  (LUT4=1 LUT5=1 LUT6=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
```

3.2 Simulation

3.2.1 Testbench 1

K = 36, ADD = 100 This testbench mainly checks the correct decrement of `reg_k`, which starts from 31 (with

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo	33	31	33	30	33	29	33	28	33	27	33	26	33	25	33	24	33	23	33	22
Addr	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139
Prima	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo	33	21	33	20	33	19	33	18	33	17	33	16	33	15	33	14	33	13	33	12
Addr	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Prima	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo	33	11	33	10	33	9	33	8	33	7	33	6	33	5	33	4	33	3	33	2
Addr	160	161	162	163	164	165	166	167	168	169	170	171								
Prima	0	0	0	0	0	0	0	0	1	1	0	1								
Dopo	33	1	33	0	33	0	33	0	1	31	1	30								

the first non-zero value 33 at address 100) and is decremented down to 0, where it remains for 3 iterations. It also shows that non-zero values at addresses intended for credibility do not cause any issues. Moreover, since this is the first testbench presented, it correctly tests the general behavior, the handling of logic based on the `i_start` and `i_rst` signals, and the correct management of the signals `o_done`, `o_mem_en`, `o_mem_we` (for which there are corresponding `assert` statements in the testbench).

3.2.2 Testbench 2

K = 7, ADD = 65522

Addr	65522	65523	65524	65525	65526	65527	65528	65529	65530	65531	65532	65533	65534	65535
Prima	0	0	0	1	66	0	66	0	66	0	0	0	65	0
Dopo	0	0	0	0	66	31	66	31	66	31	66	30	65	31

This testbench checks the case where the initial significant values are null. It also covers the particular case in which multiple consecutive non-null significant values are equal (so, even if the same value appears again at the end, the credibility remains 31). Additionally, it verifies the edge case where the end of memory is reached.

3.2.3 Testbench 3

K = 0, ADD = 100

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	33	0	33	0																
Dopo	33	0	33	0																

This testbench checks the edge case where K=0, so there should be no iterations. (In this case, the first time the FSM reaches the FINISH state, o_done is already high and it immediately transitions to the WAIT_START state, without performing any iterations).

3.2.4 Testbench 4

ADD1 = 100, K1 = 5; ADD2 = 110, K2 = 5

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	255	1	52	0	11	0	92	0	22	0	2	0	0	0	12	0	39	0	31	0
Dopo1	255	31	52	31	11	31	92	31	22	31	2	0	0	0	12	0	39	0	31	0
Dopo2	255	31	52	31	11	31	92	31	22	31	2	31	2	30	12	31	39	31	31	31

This testbench checks the correct handling of two consecutive operation sequences, without overlap of the managed memory segments. The second sequence is started without resetting the machine first.

3.2.5 Testbench 5

ADD1 = 100, K1 = 8; ADD2 = 104, K2 = 8

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo1	33	31	33	30	33	29	33	28	33	27	33	26	33	25	33	24	0	0	0	0
Dopo2	33	31	33	30	33	31	33	31	33	31	33	31	33	31	33	31	33	30	33	29

This testbench checks the correct handling of two consecutive operation sequences, with partial overlap of the managed memory segments. The second sequence is started after resetting the machine.

4 Conclusions

The proposed testbenches (which are not the only ones used during the testing phase) appear to cover all relevant cases without being redundant. Given that the architecture behaves as expected both before and after synthesis, it is reasonable to assume that it has been correctly designed and implemented.

I am aware that the proposed VHDL implementation is not the easiest to write or the most straightforward to read, as it contains several Structural parts and adders that could have been left to VHDL to implement automatically. However, I intentionally wrote the implementation this way to explore VHDL as much as possible according to the methods presented in class.