Arquiteturas de Alto Desempenho

Second practical assignment — VHDL description and simulation of an indexed accumulator

Guilherme Craveiro (103574), João Gaspar (107708)

Departamento de Eletrónica, Telecomunicações e Informática (DETI)

Universidade de Aveiro

January 4, 2025

Contents

1 Introduction

The purpose of this project is to design and implement a sequential logic circuit in VHDL that simulates the behavior of a custom accumulator. The accumulator is defined by a C function where data is stored in an array, and specific read and write operations are performed on evert clock cycle. The circuit processes arguments including write address, write increment, and read address, and returns the corresponding read data from a memory array.

This report describes the implementation of the accumulator in multiple stages, including:

1. **Single-Cycle Implementation:** Performing read, addition, and write operations within a single clock cycle.
2. **Pipelined Implementation:** Splitting the operations over two clock cycles, ensuring proper handling of consecutive write operations to the same address.
3. **Modified Accumulator:** Introducing a shift operation to the increment value using an efficient barrel shifter and integrating this feature into the accumulator.

The implementation utilizes provided VHDL entities such as *adder_n* for arithmetic operations, *dual_port_ram* for memory management and a testbench made by the professor to test the single-cycle implementation. Each design phase includes the creation of a corresponding architecture, testbench, and the determination of the smallest viable clock period to achieve functional operation. This report outlines the steps taken to complete these tasks, alongside insights gained from simulation and testing.

2 Single-Cycle Implementation

The single-cycle implementation of the accumulator performs the read, addition, and write operations within the same clock cycle. This approach ensures that the accumulator can process one set of inputs per clock cycle, making it highly efficient for applications requiring rapid data processing.

2.1 Initial Design Considerations

The provided block diagram suggested read_addr as an output signal, implying a registered output. However, our implementation uses read_addr as an input signal directly connected to the RAM's read port. This design choice was made because:

- The C function specification shows that read operations should return the current memory value without registration
- Asynchronous reads enable single-cycle operation as required
- Direct connection simplifies the design while meeting functional requirements

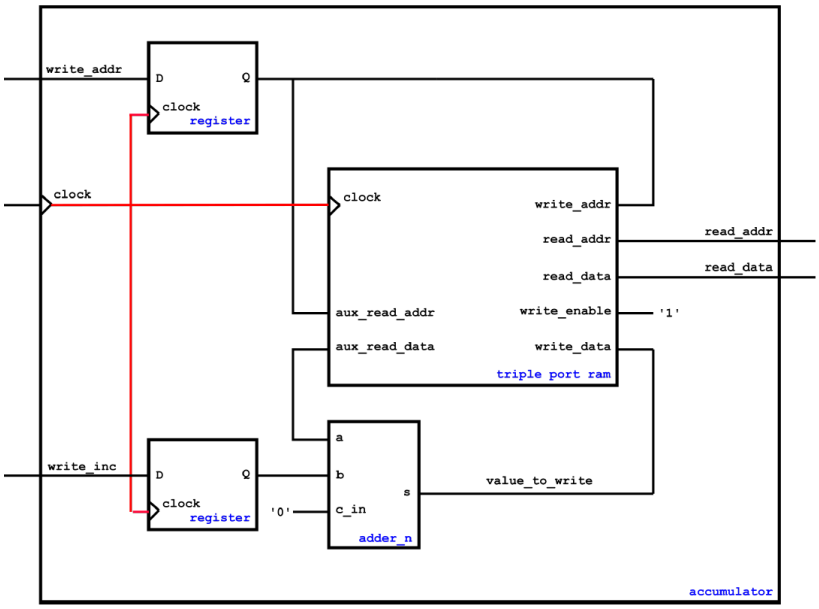

Figure 1: Block diagram showing read_addr as output, though implemented as input in final design

2.2 Dual-Port RAM

The dual_port_ram implementation underwent significant modifications to simplify its architecture while maintaining essential functionality. The key changes between the original and final implementations are detailed below.

The updated architecture was simplified to:

- Separate clock and write_en conditions in write process
- Direct asynchronous read without process declaration
- Removed transport delays
- Removed special case handling for address conflicts

```
architecture asynchronous_new of dual_port_ram is
  type ram_t is array(0 to 2**ADDR_BITS-1) of
    std_logic_vector(DATA_BITS-1 downto 0);
  signal ram : ram_t := (others => (others => '0'));
begin
  -- write sincrono
  process(clock) is
  begin
    if rising_edge(clock) then
      if write_en = '1' then
        ram(to_integer(unsigned(write_addr))) <= write_data;
      end if;
    end if;
  end process;

  -- read assincrono
  read_data <= ram(to_integer(unsigned(read_addr)));
end asynchronous_new;
```

The modifications resulted in:

- Cleaner, more maintainable code structure
- Potentially different behavior when reading and writing to the same address
- Removal of artificial timing delays, allowing synthesis tools to determine appropriate timing
- Simpler synchronization model with clear separation of synchronous writes and asynchronous reads

2.3 Accumulator Implementation

The accumulator implementation follows the block diagram structure with some modifications. The design uses two instances of the dual_port_ram and one adder_n to achieve single-cycle operation.

2.3.1 Signal and Component Organization

The key signals in the implementation are:

- s_current_value: Holds the current value read from auxiliary RAM
- s_sum: Stores the addition result

- s_carry_out: Captures overflow from addition
- write_enable: Permanently set to '1' as per requirements

2.3.2 Block Diagram Implementation

The implementation correlates with the block diagram as follows:

- Main RAM (ram):
  - Connects directly to read_addr for asynchronous reads
  - Receives s_sum as write_data
  - Shares the write_addr input
- Auxiliary RAM (aux_ram):
  - Uses write_addr for both read and write addresses
  - Shares the same write_data (s_sum) as main RAM
  - Provides s_current_value for addition
- Adder:
  - Takes s_current_value and write_inc as inputs
  - Produces s_sum for both RAMs

```
1  architecture structural of accumulator is
2    signal s_current_value : std_logic_vector(2**
       DATA_BITS_LOG2-1 downto 0);
3    signal s_sum           : std_logic_vector(2**
       DATA_BITS_LOG2-1 downto 0);
4    signal s_carry_out     : std_logic;
5    signal write_enable    : std_logic;
6  begin
7    write_enable <= '1';
8    -- Component instantiations as per block diagram
9    [...]
10 end structural;
```

2.4 Test Results

The testbench results confirm the correct operation of the accumulator through a sequence of write operations followed by verification reads. The waveform analysis shows:

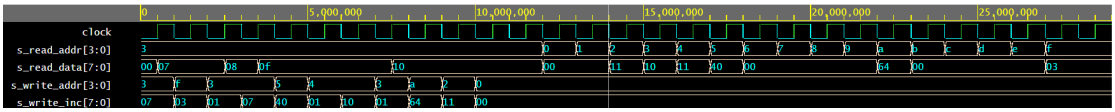

Figure 2: Results for the Single-Cycle Implementation without barrel shifter

2.4.1 Write Operations

Key write operations sequence (shown in hex → decimal):

- a[2]: Single write of 0x11 (17 in decimal)
- a[3]: 0x07 → 0x08 → 0x0f → 0x10 (7 → 8 → 16 → 16 in decimal)
- a[4]: 0x01 → 0x11 (1 → 17 in decimal)
- a[5]: Single write of 0x40 (64 in decimal)
- a[10]: Single write of 0x64 (100 in decimal)
- a[15]: Single write of 0x03 (3 in decimal)

2.4.2 Read Verification

The subsequent read sequence verified all memory positions (0-15), confirming the expected final values. The s_read_data signal showed correct values matching the write operations, with unwritten addresses containing 0 as expected.

The simulation confirms that both the accumulation operations and memory access functionality work as specified in the original requirements.

3 Single-Cycle Implementation with Barrel Shifter

The barrel shifter is a key component added to the accumulator to enable efficient shifting of the increment value. This feature allows the accumulator to perform left shifts on the increment, multiplying it by powers of 2, which is particularly useful in applications requiring scalable arithmetic operations.

3.1 Barrel Shifter Design

The barrel shifter was implemented as a structural VHDL entity. It accepts an input vector (`data_in`), a shift amount (`shift`), and produces the shifted output (`data_out`). The design supports up to 31-bit shifts and is scalable through the `DATA_BITS` generic.

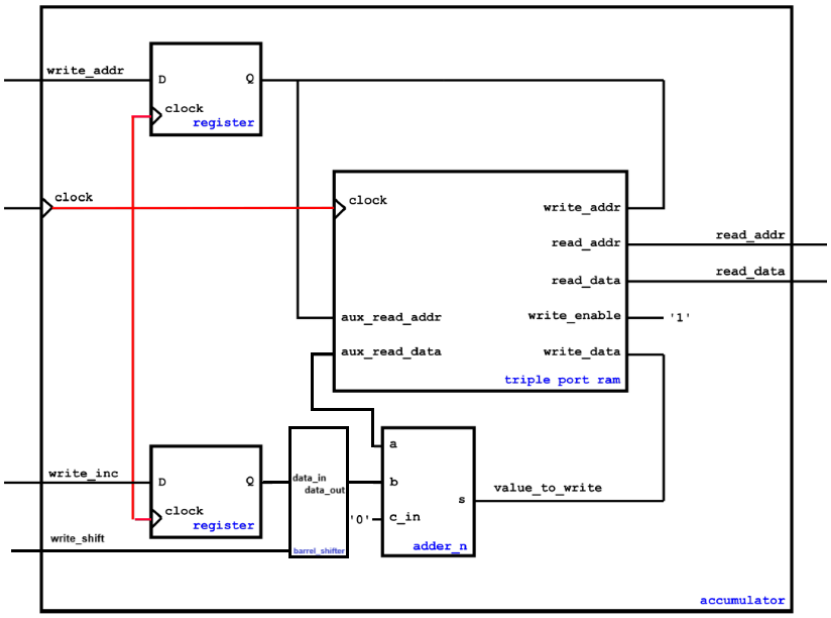

Figure 3: Block diagram showing Single-Cycle Implementation with barrel shifter

The shifter operates through a series of processes, each handling a specific shift magnitude, shift by 1 bit, 2 bits, 4 bits, 8 bits and 16 bits.

Each process conditionally shifts the input based on the corresponding bit of the `shift` signal, propagating the result to the next stage. The final output is assigned to `data_out`. This cascading design ensures an efficient and low-latency implementation.

```
-- shift de 1 bit
  shift1: process(data_in, shift(0))
  begin
    if shift(0) = '1' then
      shift1_out <= data_in(DATA_BITS-2 downto 0) & '0';
    else
      shift1_out <= data_in;
    end if;
  end process;

  -- shift de 2 bits
  shift2: process(shift1_out, shift(1))
  begin
    if shift(1) = '1' then
      shift2_out <= shift1_out(DATA_BITS-3 downto 0) & "00";
    else
```

```
17        shift2_out <= shift1_out;
18      end if;
19    end process;
```

3.2 Integration into Accumulator

To integrate the barrel shifter into the accumulator, the following steps were taken:

1. Input Extension: The `write_shift` signal from the accumulator is extended to 5 bits (`s_extended_shift`) to match the shifter's input requirements.
2. Connection to Increment Value: The shifter processes the `write_inc` signal, producing a shifted value (`s_shifted_inc`).
3. Addition with Current Value: The shifted increment is added to the current value read from the auxiliary RAM, and the result is written back to both RAMs.

```
1  architecture structural of accumulator is
2    signal s_current_value   : std_logic_vector(2**
       DATA_BITS_LOG2-1 downto 0);
3    signal s_shifted_inc     : std_logic_vector(2**
       DATA_BITS_LOG2-1 downto 0);
4    signal s_sum             : std_logic_vector(2**
       DATA_BITS_LOG2-1 downto 0);
5    signal s_carry_out       : std_logic;
6    signal s_extended_shift : std_logic_vector(4 downto 0);
       -- para o barrel shifter
7    signal write_enable      : std_logic;
8  begin
9    write_enable <= '1';
10
11   s_extended_shift <= "00" & write_shift;
12
13   shifter : entity work.barrel_shifter
14     generic map
15     (
16       DATA_BITS => 2**DATA_BITS_LOG2
17     )
18     port map
19     (
```

```
20        data_in   => write_inc,
21        shift     => s_extended_shift,
22        data_out  => s_shifted_inc
23    );
```

3.3 Simulation Results

The simulation of the accumulator with the barrel shifter confirmed its functionality:

- The shift operation modifies the `write_inc` value as expected for all shift values from 0 to 31.
- The accumulator correctly adds the shifted increment to the current memory value.
- No timing violations or functional errors were observed during the tests.

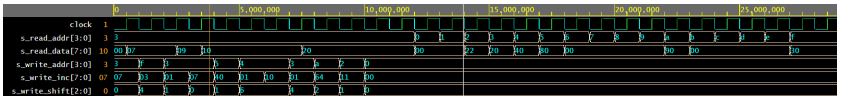

Figure 4: Simulation results showing barrel shifter functionality integrated into the accumulator.

The barrel shifter enhances the accumulator's capabilities by enabling efficient scaling of increment values through shifting. Its integration maintains the accumulator's single-cycle operation while adding flexibility to arithmetic computations. Future optimizations may focus on reducing resource usage for higher bit-widths.

4 Pipelined Implementation

The single-cycle approach ensures that all operations (read, add, write) occur within one clock cycle. While simple, this approach constrains the maximum clock frequency because it must accommodate RAM read time, addition time, and RAM write time in a single cycle.

To address this, we developed a **pipelined** (two-cycle) version of the accumulator. In this version, the read of the old value occurs in *Stage 1*, and the addition + write-back occur in *Stage 2* of the pipeline on the following clock cycle.

4.1 Design Overview

In the pipelined design:

- **Stage 1 (Cycle N):** We read the old value from an auxiliary RAM (`aux_ram`) at address `write_addr`. Simultaneously, we optionally shift the increment via a barrel shifter (if enabled in the design). Both the old value and the shifted increment are stored in pipeline registers on the rising edge of the clock.
- **Stage 2 (Cycle N+1):** The latched old value and shifted increment are fed into the adder, producing the sum. This sum is then written back to both the main RAM (`ram`) and the auxiliary RAM (`aux_ram`) at the (latched) `write_addr`. Meanwhile, the RAM is also servicing the read port (`read_addr`) in parallel.

This separation allows each stage to fit comfortably in a shorter clock period, potentially increasing the maximum operating frequency. The trade-off is that the total time from initiating a "write increment" to seeing the updated memory value is now two clock cycles (an additional latency).

4.2 Architecture Changes

Compared to the single-cycle version, the main architectural difference is:

1. **Pipeline Registers:** We introduce registers to hold:
   - The old value read from `aux_ram` (on `write_addr`)
   - The write increment
   - The `write_addr` itself

   These are captured at the end of Stage 1 and used in Stage 2.
2. **Hazard/Bypass Logic:** If the external `read_addr` matches the address being written in the current cycle's Stage 2, we bypass the new sum directly to the output, avoiding the old data from the RAM.

An outline of the pipelined accumulator's VHDL architecture is below (omitting some details for brevity):

```
1 architecture pipelined of accumulator is
2   -- Pipeline registers
3   signal p1_write_addr_reg  : std_logic_vector(...);
4   signal p1_old_value_reg   : std_logic_vector(...);
```

```
5   signal p1_write_inc_reg   : std_logic_vector(...);
6
7   -- Other signals
8   signal s_sum         : std_logic_vector(...);
9   signal s_carry_out   : std_logic;
10  signal s_ram_read_data, s_aux_read_data : std_logic_vector
      (...);
11
12 begin
13
14  aux_ram : entity work.dual_port_ram ...
15    port map(
16      write_addr => p1_write_addr_reg,
17      write_data => s_sum,
18      read_addr  => write_addr,
19      read_data  => s_aux_read_data
20    );
21
22  process(clock)
23  begin
24    if rising_edge(clock) then
25      p1_write_addr_reg  <= write_addr;
26      p1_old_value_reg   <= s_aux_read_data;
27      p1_write_inc_reg <= s_shifted_inc;
28    end if;
29  end process;
30
31  adder : entity work.adder_n ...
32    port map(
33      a => p1_old_value_reg,
34      b => p1_write_inc_reg,
35      ...
36      s => s_sum
37    );
38
39  ram : entity work.dual_port_ram ...
40    port map(
41      write_addr => p1_write_addr_reg,
42      write_data => s_sum,
43      ...
44      read_addr  => read_addr,
45      read_data  => s_ram_read_data
46    );
```

```
47
48    -- Bypass if read_addr = p1_write_addr_reg
49    process(all)
50    begin
51      if (read_addr = p1_write_addr_reg) then
52        read_data <= s_sum;
53      else
54        read_data <= s_ram_read_data;
55      end if;
56    end process;
57
58  end pipelined;
```

4.3 Simulation and Results

We reused the same test bench stimulus as in the single-cycle case but **added a brief extra delay** after the last write operations before reading out the entire memory. This ensures the pipeline has fully updated the memory location in Stage 2.

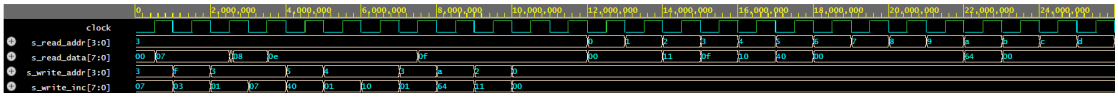

Figure 5: Simulation waveforms for pipelined accumulator. Notice the one-cycle delay between write address/increment and the final updated memory value.

Observations:

- The final memory contents match the expected values from the test bench, verifying correct read–modify–write behavior over two cycles.
- Comparing to the single-cycle version, each `write_addr`/ `write_inc` update takes one extra clock cycle to appear in memory. This is the inherent latency of a two-stage pipeline.
- Bypass logic ensures that if `read_addr` coincides with the address being updated, we output the new sum immediately.

4.4 Conclusion

By splitting the read–modify–write sequence into two stages, we reduced the timing constraints within each cycle. This approach offers potential for a higher

clock frequency at the cost of an additional cycle of latency. All functional requirements are met, including handling consecutive writes to the same address (courtesy of bypass logic or "write-first" RAM mode).

This design can be further extended or deepened (e.g., multi-stage pipelines) if higher performance or additional operations are required in future designs.

5 Pipelined Implementation with Barrel Shifter

Following the successful integration of a barrel shifter in the single-cycle accumulator, we proceeded to create a **two-cycle pipelined** version that also incorporates shifting. This design separates the read and shift operations (Stage 1) from the addition and write operations (Stage 2), thereby relieving timing pressure within one clock cycle and potentially allowing higher operating frequencies.

5.1 Rationale and Overview

In the single-cycle design, reading from RAM, shifting the increment, and adding it to the old value must all happen before the next rising clock edge. This can restrict the maximum clock frequency, especially as bit widths increase.

By splitting the operation into two stages, the design allows:

- **Stage 1:**
  - Read the old value from an auxiliary RAM (`aux_ram`) using the current `write_addr`.
  - Shift the increment (`write_inc`) by `write_shift` using the barrel shifter.
  - Store both (old value + shifted increment) in pipeline registers at the clock edge.
- **Stage 2:**
  - Add the latched old value with the latched shifted increment in the adder.
  - Write the result (`s_sum`) back to both `ram` and `aux_ram` at the latched `write_addr`.

This pipeline architecture introduces a one-cycle latency (from increment input to memory update) but can run at a higher clock rate due to reduced per-cycle work.

5.2 Pipeline Architecture and Key Signals

Stage 1

- **aux_ram** is read at `write_addr`.
- The barrel shifter shifts `write_inc` according to `write_shift`.
- Pipeline registers `p1_old_value_reg` and `p1_shifted_inc_reg` capture these outputs on the rising clock edge.

Stage 2

- The adder adds `p1_old_value_reg` to `p1_shifted_inc_reg`.
- The result `s_sum` is written back to both `ram` and `aux_ram` at `p1_write_addr_reg`.

For the external `read_addr`, the main `ram` is read asynchronously in parallel. If `read_addr` happens to match `p1_write_addr_reg` in the same cycle, we bypass the new sum (`s_sum`) to `read_data`, ensuring we see the updated value immediately.

5.3 VHDL Implementation Highlights

A simplified version of our pipelined VHDL code is shown below, focusing on the new or changed portions. The remaining modules (`dual_port_ram`, `adder_n`, and `barrel_shifter`) remain unchanged in functionality.

```
architecture pipelined of accumulator is
 ...

begin

  s_extended_shift <= "00" & write_shift;

  shifter: entity work.barrel_shifter
    generic map(
      DATA_BITS => 2**DATA_BITS_LOG2
    )
    port map(
      data_in  => write_inc,
      shift    => s_extended_shift,
      data_out => s_shifted_inc
    );

...
```

```
19
20    process(clock)
21    begin
22      if rising_edge(clock) then
23        p1_write_addr_reg   <= write_addr;
24        p1_old_value_reg    <= s_aux_read_data;
25        p1_shifted_inc_reg  <= s_shifted_inc;
26      end if;
27    end process;
28
29    adder: entity work.adder_n
30      generic map(
31        N => 2**DATA_BITS_LOG2
32      )
33      port map(
34        a      => p1_old_value_reg,
35        b      => p1_shifted_inc_reg,
36        c_in   => '0',
37        s      => s_sum,
38        c_out  => open
39      );
40
41  ...
42
43  end pipelined;
```

5.4 Simulation and Results

We reused our existing test bench stimuli, which writes various values (with shifts) to memory addresses and then reads them back. To account for the pipeline's extra clock cycle, we inserted additional idle cycles after the final write before reading out the memory content. This ensures the pipeline has completed its last Stage 2 write.

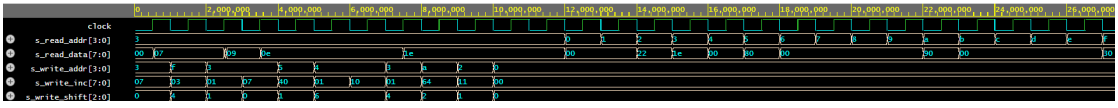

Figure 6: Simulation waveforms for the pipelined accumulator with barrel shifter. Note the one-cycle delay between the shift + read (Stage 1) and the addition + write (Stage 2).

Observations:

- Each `write_addr` and `write_shift` change triggers a shifted increment `s_shifted_inc` in the same cycle. However, the final sum is only written to memory on the subsequent clock cycle.
- Consecutive writes to the same address are handled correctly by either RAM's write-first mode or by the explicit bypass logic shown above.
- The final memory contents (after the extra wait cycles) match the expected results, confirming correct read–modify–write behavior with shifting.

5.5 Conclusion

The pipeline approach, when extended to incorporate shifting operations, provides a balance between throughput and clock frequency. By splitting the read-/shift and add/write steps into two distinct stages, the design can meet more stringent timing requirements. Meanwhile, the barrel shifter offers flexible scaling of the increment value at minimal additional latency. Overall, this *pipelined accumulator with barrel shifter* meets the functional requirements while enhancing performance potential for higher-speed operation.

6 Conclusion

The project successfully implemented an indexed accumulator in VHDL, fulfilling all functional requirements. Through simulation and testing, we verified the single-cycle and pipelined implementations, as well as the integration of the barrel shifter.

7 Autoevaluation

In this project, both group members contributed significantly, dividing the tasks evenly.

- Guilherme Craveiro: 50% of the work.
- João Gaspar: 50% of the work.