Computer Architecture Project

1401.2 Semester AUT

Author: Seyyed Mohammad Hamidi
June 1, 2023

Abstract

In a nutshell, the Finite State Machine (FSM) is a sequential circuit that regulates the sequence of operations in digital systems. This FSM presented in the code has four states: Fetch, Decode, Read Signal, and Execute.
The code begins by defining the states and initializing some signals to control the FSM. The signals include PC (Program Counter), opcode, address, ALU_clk (ALU clock), RAM_clk (RAM clock), ROM_clk (ROM clock), and a few more.
The FSM then defines the components of the ALU, RAM, and ROM modules and establishes connections between the FSM and these modules using port maps.
In the main process, the FSM operates based on the clock signal (CLK) and the reset signal. It starts in the Fetch state, where it fetches the instructions from ROM and increments the PC. Then, in the Decode state, it decodes the instruction fetched. The opcode and the address of the operand are extracted.
In the read_signal state, the RAM module is used to fetch the operand from the memory location specified by the address. After that, in the Execute state, the ALU executes the operation specified by the opcode using the operand and the value in the accumulator (AC).

Report on VHDL Module: FSM

1.1 Overview

The given VHDL module is designed as a Finite State Machine (FSM). The module is named `FSM` and contains components `ALU`, `RAM`, and `ROM` to simulate the operation of a simple computer. This FSM module has four states: Fetch, Decode, Read, and Execute. These states represent the steps of an instruction cycle of a computer.

The FSM module is sensitive to the clock signal (`clk`) and reset signal (`reset`). If the reset signal is set to '1', the FSM will return to the initial Fetch state and reset the program counter (`PC`). The FSM's done signal (`FSM_done`) is set to '1' at the end of the process execution, indicating that the FSM has finished a cycle of operation.

1.2 FSM Architecture

The FSM architecture is described in the `Behavioral` section. It defines several signals, components, and their corresponding port maps, along with the main process that operates the state transitions.

1.2.1 Signals

The architecture defines several signals including state signal, accumulator `AC`, data register `DR`, instruction register `IR`, carry flag `E`, program counter `PC`, opcode, address, clock signals for ALU, RAM, and ROM, start signal `init_start`, and done signals for each component. These signals serve as the internal data and control paths of the FSM.

1.2.2 Components

There are three components defined: `ALU`, `RAM`, and `ROM`, each representing the Arithmetic Logic Unit, Random Access Memory, and Read-Only Memory of a simple computer, respectively.

1.2.3 Port Maps

Port maps are used to establish the interface connections between the components and the signals. Each component has its inputs and outputs mapped to the corresponding signals.

1.3. Main Process and State Transitions

The main process of the architecture is sensitive to the clock and reset signals. At each rising edge of the clock, if the reset signal is '1', the FSM goes to the initial Fetch state, and the program counter is reset to zero. Otherwise, depending on the current state, the FSM performs the corresponding operation and transits to the next state.

1. `Fetch` state: The program counter is incremented and an instruction is fetched from ROM. When the instruction is fetched successfully, the state transitions to Decode.

2. `Decode` state: The opcode and address parts of the instruction are extracted. After the decoding process, the state transitions to the Read state.

3. `Read` state: Depending on the opcode, if the operation requires writing to RAM, the `write_in` signal is set to '1'. A read operation from RAM is needed if the address is not zero. After the reading process is done, the state transitions to Execute.

4. `Execute` state: The ALU is called to execute the operation specified by the opcode. If the operation increments the PC, the PC is incremented. After the execution is done, the state transitions back to the Fetch state.

At the end of each cycle of the FSM operation, the FSM_done signal is set to '1'.

1.4. Code Explanation

This FSM module models a simple computer operation using VHDL. The computer has an ALU for performing arithmetic and logical operations, RAM for read-write memory operations, and a ROM for storing the program. The FSM operates in a cycle of Fetch-Decode-Read-Execute states, modeling the instruction cycle of a computer. In each cycle, the FSM fetches an instruction from ROM, decodes the instruction into opcode and address, reads data from RAM if needed, and executes the operation using ALU.

The code

uses the standard IEEE libraries `std_logic_1164` for the logic operations, `numeric_std` for the numeric operations, and `STD_LOGIC_UNSIGNED` for the unsigned arithmetic operations. It also uses the component modeling method to define the components of ALU, RAM, and ROM and uses the port map method to connect the components and signals.

The process in the code is sensitive to the rising edge of the clock and reset signals, making it a synchronous sequential circuit. The state transition and operation are controlled by the clock signal, while the reset signal is used to reset the FSM to the initial state. The signals and components are designed to be as generic as possible, so they can be easily customized for different specifications.

Report on VHDL Module: ALU

2.1. overview

The given VHDL (VHSIC Hardware Description Language) code defines an Arithmetic Logic Unit (ALU) using a standard design approach. ALU is a key component of a computer's central processing unit (CPU). It performs arithmetic and bitwise operations on binary numbers. The design is constructed using a combination of VHDL libraries such as `ieee.std_logic_1164`, `IEEE.numeric_std`, and `IEEE.STD_LOGIC_UNSIGNED`.

The ALU module defined in the code supports various operations like AND, store, load, add, increment, clear, shift, compare and multiply, each of which is selected based on a 6-bit opcode input. It also implements a square root operation which is part of a custom function package, `Functions_package`.

2.2. Code Overview

The ALU module code is divided into the following main parts:

2.2.1. Functions Package Definition (`Functions_package`)

The code begins with the definition of a package called `Functions_package`. This package contains two functions - `multiply` and `SQR`.

- The `multiply` function multiplies two 16-bit numbers using the classic multiplication algorithm, where each bit of the multiplier is used to conditionally add the multiplicand to a running total.

- The `SQR` function calculates the square root of a number using an iterative method.

2.2.2. ALU Entity Definition

The entity declaration for the ALU module is defined with inputs, outputs, and signals. The main inputs include two 16-bit data inputs (`AC` and `DR`), a 6-bit opcode for function selection (`opcode`), and a clock signal (`ALU_clk`). The outputs include a 16-bit result output (`ALU_out`), a carry-out flag (`E_out`), a program counter increment signal (`inc_PC`), and an operation complete signal (`done`).

2.2.3. ALU Behavioral Architecture

The main body of the ALU module is defined under the architecture section. The behavior of the ALU is described within a clocked process that executes on the rising edge of the `ALU_clk` signal. Depending on the opcode input, the ALU performs various arithmetic and logic operations.

2.3. Code Explanation

2.3.1. Functions Package (`Functions_package`)

- The `multiply` function takes two 16-bit `std_logic_vector` inputs. The multiplication is performed by initializing all variables, performing a loop 16 times, and returning the lower 16 bits of the result. The function performs bit-level manipulations on the inputs, utilizing the two's complement method for subtraction.

- The `SQR` function takes an `unsigned` input and returns an `unsigned` output. This function calculates the square root of the input using a loop and bit-level manipulations. It performs iterative additions and subtractions, adjusting the result based on whether a subtraction overflows.

2.3.2. ALU Behavioral Architecture

The main process in the ALU module's architecture waits for the rising edge of the clock signal `ALU_clk`. If `ALU_needed` is high, then depending on the `opcode`, the ALU performs a corresponding operation on the inputs `AC` and `DR`.

- For instance, opcode `"000001"` executes an AND operation on `AC` and `DR`, and opcode `"000010"` executes a store operation, storing the value of `AC` in `ALU_result`.

- Opcodes `"000100"` and `"010000"` call the `multiply` and `SQR` functions from the `Functions_package`, respectively.

- Opcodes `"001010"` to `"

001101"` implement conditional increment of the program counter based on specific conditions.

- Some opcodes like `"001000"` and `"001001"` perform bit shift operations on the `AC` input.

- An overflow is checked after each operation, with the carry-out flag `E_out` set based on the most significant bit of the temporary result `tmp`.

- Finally, the `done` signal is set to `'1'` to indicate the completion of the ALU operation.

2.4. Conclusion

The given VHDL code is an efficient implementation of an ALU module. The functionality provided by the code is extensive, ranging from basic bitwise operations to complex functions such as multiplication and square root calculations. The code is well-structured and provides a clear insight into the design and operation of an ALU in a digital computer system.

Report on VHDL Module: RAM

3.1. Overview

This report provides a comprehensive overview of the provided VHDL RAM module code, detailing its functionality, structure, and essential components. This RAM (Random Access Memory) module is designed for a digital system using the VHDL (VHSIC Hardware Description Language) for its design and development. The module includes various input and output ports to support data writing, reading, and addressing, as well as synchronous operations with a system clock.

3.2. Code Description

The given VHDL RAM module uses various libraries:

- `IEEE.STD_LOGIC_1164.ALL`: This library provides the essential declarations for multilevel signal modeling.

- `IEEE.STD_LOGIC_UNSIGNED.ALL`: It offers functionalities for unsigned std_logic_vector types.
- `IEEE.numeric_std.ALL`: This library is used for numeric operations.

3.2.1. Entity Description

The entity named `RAM` defines the interface of the module. It consists of the following ports:

- `address`: An input signal of 10 bits (9 downto 0), which defines the address line for the memory module.
- `data_in`: An input data line of 16 bits (15 downto 0), used to write data into the memory.
- `write_in`: An input signal to control the write operation. When this signal is high ('1'), the module performs a write operation.
- `RAM_clk`: An input clock signal, which controls the synchronization of the operations.
- `data_out`: An output data line of 16 bits, used to read data from the memory.
- `RAM_needed`: An input signal to indicate when the RAM module operation is needed.
- `done`: An output signal to indicate the completion of the memory operation.

3.2.2. Architecture Description

The architecture named `Memory` of the RAM entity comprises the following elements:

- `RAM_Array`: This is a local type declaration, creating an array type of 1024 elements (0 to 1023), each a std_logic_vector of 16 bits. It represents the memory storage of the RAM module.
- `RAM_data`: This is a signal of the `RAM_Array` type, which is initialized to "0000000000000000". It serves as the main memory storage of the module.

- `done_signal`: This is a signal used internally to control the `done` output port.

3.2.2.1. Process Description

The main process named `main` is sensitive to the `RAM_clk` signal. At the rising edge of the clock signal, if `RAM_needed` is high ('1'), the process checks the `write_in` signal:

- If `write_in` is high ('1'), the data from `data_in` is written to the `RAM_data` at the location specified by `address`.
- If `write_in` is low ('0'), the data from `RAM_data` at the location specified by `address` is put on the `data_out` line.

The `done_signal` is set to '1' after the operations, indicating the completion of the process.

3.3 Code Functionality

This VHDL RAM module is a simplified representation of a RAM system, allowing for reading and writing operations at any memory address, as specified by the `address` input. The module works synchronously with the rising edge of the input clock signal `RAM_clk`. The `write_in` signal controls whether the module will write to or read from memory. If `write_in` is '1', the module writes the `data_in` into the memory location specified by `address`. If `write_in` is '0', the module reads the data from the memory location specified by `address` and sends it to `data_out`.

The `done` signal is set high after each clock cycle where

the RAM operation was needed, indicating that the current operation (read or write) is complete. This can be used by other parts of the system to know when data has been successfully written or read.

The design is a simple synchronous system and does not handle any error conditions such as invalid addresses, writing without needing RAM, or simultaneous read-and-write requests.

3.4 Conclusion

This VHDL RAM module is a fundamental design of a RAM storage system used in digital systems. It showcases the use of synchronous system design, conditional data reading/writing, and address decoding. However, real-world RAM modules can be far more complex and often include features like error correction, simultaneous read and write capabilities, and various optimizations for speed and efficiency.

Report on VHDL Module: ROM

4.1 Overview

This report presents a detailed review of a VHDL (Very High-Speed Integrated Circuits Hardware Description Language) code for a ROM (Read-Only Memory) module. VHDL is a versatile and powerful hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

4.2 Code Overview

The provided code describes a VHDL model for a ROM module with a 10-bit address line and a 16-bit data output. It means this ROM module can store 2^10 (1024) instructions each of 16 bits. The ROM module is synchronous with the system clock, as the operation of reading from the ROM occurs at the rising edge of the clock signal.

4.3 Ports Description

The VHDL code contains the entity named ROM with five ports:

1. `ROM_clk` (Input, std_logic): The clock signal, which controls the reading operation of the ROM.

2. `address` (Input, std_logic_vector): The address lines for the ROM, are used to select the particular memory location. The width of the address vector is 10 bits allowing for 1024 memory locations (2^10).

3. `inst_out` (Output, std_logic_vector): The 16-bit instruction output from the selected ROM address.

4. `ROM_needed` (Input, std_logic): A control signal that triggers the ROM to provide the output to the `inst_out` when set to '1'.

5. `done` (Output, std_logic): A signal that indicates the completion of the read operation from ROM.

4.4 Internal Architecture

The ROM's internal architecture, called 'Instructions', describes the behavior of the ROM module.

The memory of the ROM module is declared with a constant `our_ROM` which is an array of 1024 elements, each element being a 16-bit std_logic_vector. Initially, all memory locations are initialized to "0000000000000000".

An internal signal `done_signal` is used to indicate the start and end of a read operation.

A process called `main` is sensitive to the `ROM_clk` signal. Whenever there's a rising edge on `ROM_clk`, if `ROM_needed` is '1', the content from the `our_ROM` at the `address` location is read and placed on `inst_out`. Once this operation is complete, the `done_signal` is set to '1'.

4.5 Code Explanation

The ROM module is operated by a clock signal `ROM_clk`. At the rising edge of this clock, if `ROM_needed` is high, an instruction is read from the ROM's memory array `our_ROM` at the location specified by the `address` input.

The address is first converted to an integer using `to_integer(unsigned(address))`. The `unsigned` function is used to interpret the `std_logic_vector` address as an unsigned integer, and the `to_integer` function converts this to a natural integer. This integer is used as the index to read the value from the `our_ROM` array.

The value read from the `our_ROM` is then assigned to `inst_out`, which is the output port of the ROM module. The `done_signal` is set to '1' at the end of each read operation, indicating the end of the process.

4.6 Conclusion

In summary, the given VHDL code successfully describes a synchronous ROM module with 1024 16-bit memory locations. The ROM module is controlled by the system clock and provides data on the output when the `ROM_needed` signal is set to '1'. The module also provides a `done` signal to indicate the end of the reading operation. It's an

efficient and simple design for a ROM module that can be utilized in various digital systems. However, to fully utilize this ROM module, instructions need to be loaded into the ROM's memory `our_ROM`.