Technical University of Crete

Single Cycle CPU

ACE312

Chamalakis Thomas - 2018030096
Stavrou Odysseas - 2018030199
April 2020

Project's Purpose:

This project (and the Phase II of it) is a substitute of the labs exercises to be taught at TUC. The projects consist of 2 phases:

1.) The Design and Implementation of the 2 essential CPU blocks (Control & DataPath), a memory module and afterwards the creation of a Single Cycle CPU which houses all 3 mentioned modules.
2.) Phase II is the creation of a multicycle CPU with the addition of pipelining.

Workflow:

Prerequired Knowledge:

- Being familiar with the Xilinx Synthesis Tools (ISE and/or Vivado) and the use of their libraries **numeric_std** and **std_logic_1164.**
- Configuring the iSim simulator with the .wcfg files to include/exclude signals.
- Core VHDL knowledge and the ability to design based on a given Block Diagram. (Acquired in ACE203 Course).

Used tools:

- Xilinx Vivado 2019.2
- Visual Studio Code with the VHDL extension
- GitHub

Design flow:

The project was announced in different Stages (I, II, III) each of them had specific objectives ie: the design of the ALU, the basic stages of the DATAPATH etc. All designed specifications were specified by the "CHARISA" (CHAnia Risc Instruction Set Architecture).

Stage I:

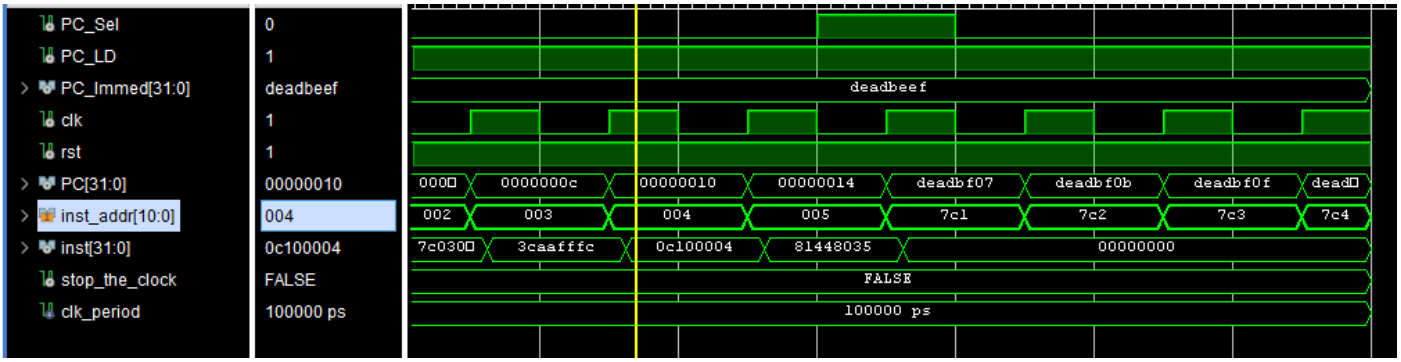
Stage I had the task of designing 2 modules: 1.) the ALU module responsible for asynchronously computing the arithmetic or logic result of two operands (or just one) depending on an input signal defining the operation. 2.) the module which contains 32, 32-bit registers and the logic to select 2 specifics ones, and 1 to write to it all based on input signals.

Note: All registers, and RAM writes happen on the positive edge of the Clk Signal.

Stage II:

For Stage II we had to design the 4 basic Stages of the DataPath:

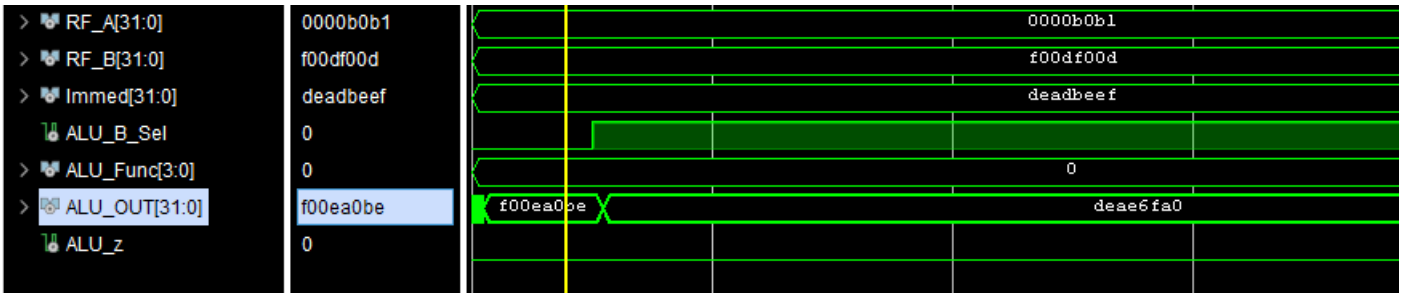
1.) The IFSTAGE:
    - Consists of 1 32-bit Register called PC, and a multiplexer.
    - When an instruction is fetched it is automatically and asynchronously determined if the branch will be taken. Because the ALU acts asynchronously it outputs immediately a flag if the last operation conducted was equal to 0. Then the multiplexer inside the stage selects the next instruction (PC + 4) or the modified one depending on the instruction executed. If the instruction is an unconditional jump then the modified PC is selected.
    - Because the memory has 2048 addresses, we map only the 11 bits between 12 and 2 of the PC address to the ram so every increment of the PC (+4) we get the instruction on the next address and not the one 4 lines below.



As we are counting + 4 for every clock cycle we fetch a new instruction from each next memory address (mapped only (12 downto 2) mentioned above). We add offset 0xdeafbeef to the register and jump to a new address far away.

2.) The EXSTAGE:

- Consists of the ALU and a simple 2:1 multiplexer for selecting the 2nd operand of the operation that is, either a register if the instruction is R-Type or the immediate if the instruction is I-Type
- The output goes to the MEMSTAGE (in case it refers to an address in memory) and back to the DECSTAGE (which contains the registers) in order to be written to the proper register.



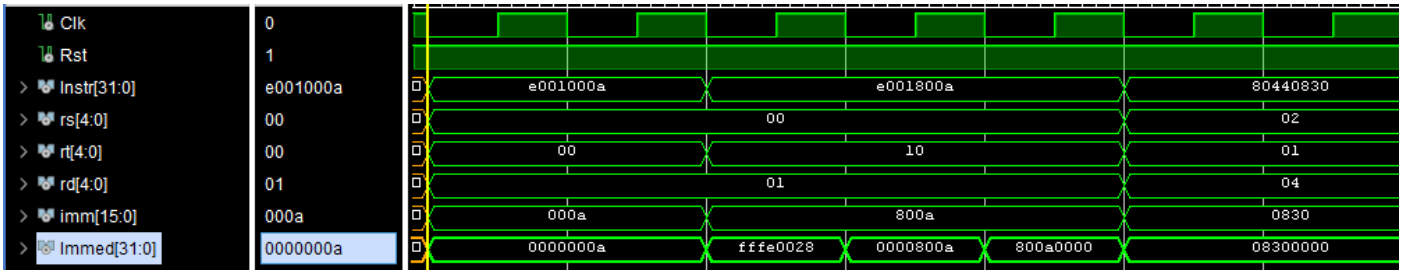
For a Simple operation (addition) we try adding operand A with operand B and with the immediate as well.

3.) The MEMSTAGE:

- A simple “interface” with the RAM which resides outside of the DATAPATH
- Takes care of the lb, sb operations because of the byte addressable ram (writes 4-byte words).
- The lb, sb instructions are both unsigned operations (from ISE: Zerofill(31 downto 0))
- Passes through, to the RAM module, the Write Enable Signal.
- An incrementor adds an 0x400 offset to the address in order to write to the data section of the memory and not onto existing instructions.
- The data that we are writing into the RAM is always the 2nd Register from the Register File.
- The memory has 2048 addresses so we map only the 10 bits between 12 and 2 of the ALU address to the ram.

4.) The DECSTAGE:

- Responsible for decoding the Instructions to the 4 required fields as soon as they come in.
- Regardless of the type of the Instruction the Immed and Rt fields are populated regardless of usage (in any case the Rt register will not be selected for the ALU operation as mentioned in the EXSTAGE).
- Contains the register file with all the registers.
- Extends the Immed depending on the instruction based on a signal.
- Using 2 multiplexers it decides, with the help of signals, which 1.) register will be the 2nd output and 2.) what is going to be the 32 bit value to be written. Either the result of the ALU or the Data we pulled from RAM.



As soon as an instruction enters the DECSTAGE it gets decoded to the 4 required fields (only the necessary fields continue onwards, ie no need for Rt in I-Type instructions etc). The 4 Immediate extensions operations are showcased: a. Sign extension b. Sign extent and << 2 c. Zero fill the 16 upper bits d. shift left by 16 and Zero fill the 16 down most bits

Stage III:

The design of the CONTROL Module and the assembly of the DATAPATH and the integration with the RAM.

1.) The CONTROL:

Throughout the report there were mentions of many signals deciding the functionality of each stage of the DATAPATH and permitting writes to the RAM. These signals originate from the CONTROL module and they are generated as soon as the new Instruction hits this module. More analiticaly the purpose of each signal is as such:

SIGNAL	PURPOSE
ALU_z	The only input signal to the control used for bne and beq instructions.
ALU_Func	Decides ALU operation: Last 4 bits of the instruction if the opcode is R-Type, or depending of the I-Type select an appropriate operation (or for ori, nand for nandi, etc.)
PC_Sel	Selects the normal increment of the PC or the modified one (depends on ALU_z)
RF_WD_Sel	Decides which 32-bit value (RAM output or ALU output) is to be written to the destination register
RF_B_Sel	Selects the 2nd Register for the Register File, for memory instruction.
RF_WE	Write Enable Signal for writing to the register, any instruction that needs to write to a register.
ImmExt (2-bit)	Extends the Immediate field using 1 of 4 different operations
ALU_B_Sel	Selects 2nd operand for the ALU, for immediate instructions.
ByteOp	Tells the MEMSTAGE that the operation it is about a single byte. Enabled only on instructions referring to single bytes (lb, sb)
MEM_WE	Write Enable Signal for writing to memory. Enabled only when on store instructions(sw, sb)

2.) The Memory:

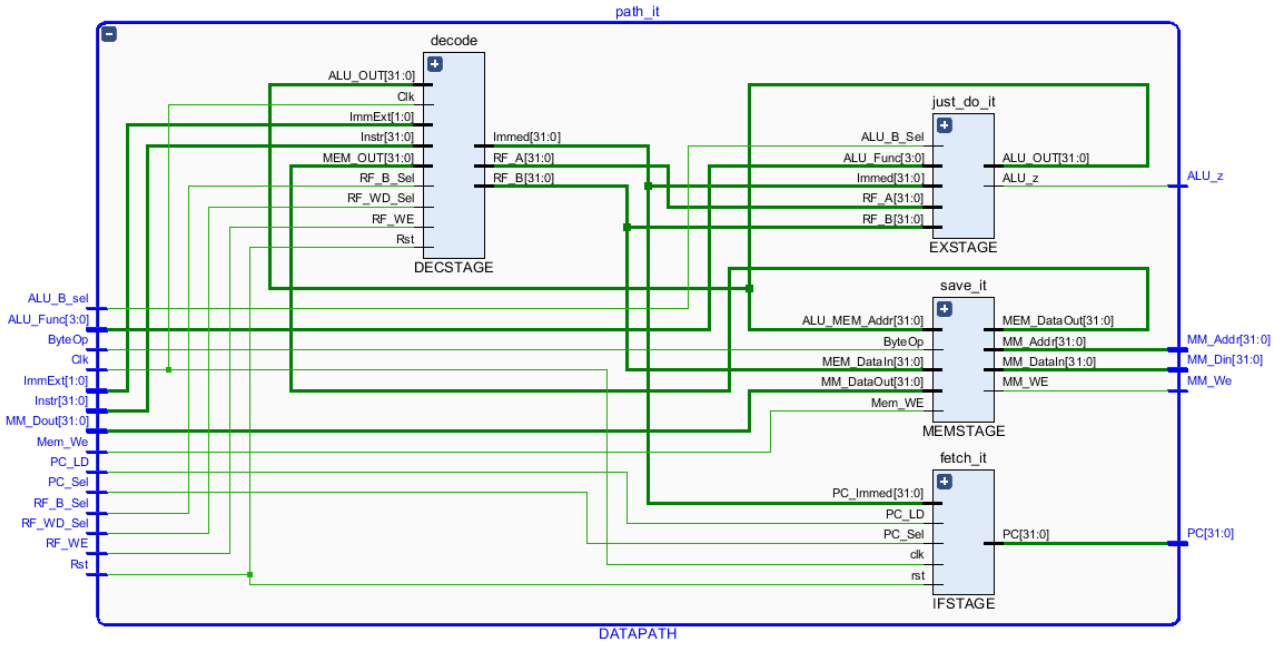
The memory Module contains the instructions in addresses 0 -> 1023 and the data in addresses 1024 -> 2043. When instantiated the module automatically populates the instructions section with the instructions present in a file called “rom.data” and also zeros everything in the data section. It synchronously writes on the rising edge of the clock signal and asynchronously reads the data and instruction values on 2 different outputs.

3.) The DATAPATH:

- Maps all 4 stages mentioned in Stage II and creates the “Path” for the data to follow depending on the instruction.
- It takes as Input the Signals coming from the Control and routes them to their proper destinations.
- Connects to the RAM module for reading/writing data and fetching the instructions.

Below is the Block Diagram representation taken from Vivado’s RTL Analysis Tools. It is clearly depicted how the 4 stages interconnect with each other and the signals going in and out of each module (signals on the left side are inputs while on the right side are outputs).

Each stage takes care of its own operations decided by the CONTROL input signals.



4.) PROC_SC:

- The top module that has only 2 inputs the clock and reset signals.
- It connects all 3 above mentioned Modules together to enable full functionality.

Design Methodology:

We designed each submodule of every DATAPATH stage independently and tested its functionality before implementing it to the upper module. As soon as every stage was completed, we tested it to ensure proper connectivity and functionality between the submodules. After completing and assembling the DATAPATH we set out to test the flow of data inside it. Acting as the control we passed the control signals for a specific instruction (already preset) and checked the inner signals if they were outputting the correct data. Either that data was the immediate extended properly, the correct register was selected, proper arithmetic operations, etc.

After ensuring the DATAPATH was functioning properly we went on testing the CONTROL. We supplied a signal and then observe if the correct signals would be pulled to HIGH and LOW accordingly.

Testing the RAM was simple enough. Just supply instruction addresses and check whether the correct instruction would be picked and also check whether we can read/write to different parts of the data section.

All reset signals are active **LOW.** On the **PROC_SC** we give a reset signal for 4-5 cycles and then we pulsed the clock for 1 clock cycle (100ns) and observed each instruction being executed.

After testing the processor with the giver programs we tested against newly generated programs (charis_3, charis_4, see README.pdf)

RESULTS & CONCLUSION:

- By analysing the ISE, we can easily figure out core specifications: number of registers, Instruction decoding etc.
- Splitting the project into small more feasible tasks will help you keep a clear mind and be more focused.
- Reuse already made components to avoid heavy and complicated code. For example, the addi instruction is the same function for the ALU with the difference being the origin of the 2nd Operand.
- The CPU is a very, very, simple component, so simple that is really difficult to grasp its usage.
- Processes are unnecessary when you don't have a periodic signal, like clock. The module can be constructed only with "when, else" statements.
- Due to the transition from ISE to Vivado, the only library needed for arithmetic purposes is the **"numeric_std".** (Had Issues while using the ISE with **numeric.signed/unsiged/arith**)
- For detecting a clock pulse, **rising/falling_edge()** is better that **wait for clk'event and clk='1' / if clk'event and clk='1'**, because **clk'event** includes every event that happens on the clock signal including U -> 1/0 which might have unwanted results.

During the first Phase of the project we learned how a simple Single Cycle CPU functions and what are the core modules responsible for its functionality. We learned how the instructions are fetched, decoded, and executed using hardware. We understood how the CPU takes decisions like when to take a branch or not and how these decisions came to be. The block diagram closely resembles the von Neumann architecture minus the external input/output devices.