Christopher Lall

CSc 342/343 – Professor Gertner

Single Cycle CPU lab

Due 5/22/2022

Table of Contents

[Objective: 3](#_Toc104124184)

[Code: 4](#_Toc104124185)

[Block Diagram & Simulation: 22](#_Toc104124186)

[Adder: 23](#_Toc104124187)

[2:1 Mux: 23](#_Toc104124188)

[Program counter: 24](#_Toc104124189)

[5bit Mux: 24](#_Toc104124190)

[Next Address Logic: 25](#_Toc104124191)

[Instruction Register: 26](#_Toc104124192)

[Instruction Memory: 26](#_Toc104124193)

[Register File: 27](#_Toc104124194)

[Data Memory: 27](#_Toc104124195)

[ALU: 28](#_Toc104124196)

[Sign Extender: 29](#_Toc104124197)

[Overflow: 29](#_Toc104124198)

[CPU Controller: 29](#_Toc104124199)

[Single Cycle CPU: 30](#_Toc104124200)

[Component Package: 30](#_Toc104124201)

[Single Cycle CPU: 31](#_Toc104124202)

[ADDI, ORI, ADDIU, NOR, LW Instructions: 31](#_Toc104124203)

[LW, ADD, ADU, SW Instructions: 32](#_Toc104124204)

[BEQ, BNE, J, SUB: 34](#_Toc104124205)

[Ultimate Test: 34](#_Toc104124206)

[Mips Summation: 36](#_Toc104124207)

[Conclusion: 37](#_Toc104124208)

# Objective:

The goal of this lab is to create a Single Cycle CPU. It will get the sum of five int’s using MIPS instructions. This is a MIPS processor with lower capabilities. All of the essential components for a MIPS CPU are coded using the VHDL language and Quartus prime. W used VHDL to program various MIPS instructions in MIPS ARMS which allowed us to manipulate instructions. It allows us to demonstrate how and what we have worked with in MIPS. In the upcoming sections, you will see vhdl code of our programs that help us as they are components we need to create the CPU. Through simulation and modules and components, Quartus Prime will help us create a design that allows us to check the input and output the sum.

Diagram, schematic

Description automatically generated

Figure 1. Single Cycle CPU schematic

# Code:

In this section I will only post the VHDL code. I will create another section to demonstrate simulation waveforms from MODELSIM. This code is in no particular order.

Text

Description automatically generated

Figure 2. VHDL code for adder

Text

Description automatically generated

Figure 3. VHDL code for 32bit 2:1 MUX

Text

Description automatically generated

Figure 4. VHDL code program counter.

Text

Description automatically generated

Figure 5. VHDL code 5bit 2:1 Mux

Text

Description automatically generated

Figure 6. VHDL code Next Address Logic

Text

Description automatically generated

Figure 7. VHDL code instruction register.

Text

Description automatically generated

Figure 8. VHDL code Instruction memory

Text

Description automatically generated

Figure 9. VHDL code register file.

Text

Description automatically generated

Figure 10. VHDL code data memory.

Text

Description automatically generated

Text

Description automatically generated with medium confidence

Figure 11. VHDL code ALU

Text

Description automatically generated

Figure 12. VHDL code extender.

Text

Description automatically generated

Figure 13. VHDL code overflow

Text

Description automatically generated

Figure 14. VHDL code CPU controller pt 1

A picture containing graphical user interface

Description automatically generated

Figure 15. VHDL code CPU controller pt 2

A picture containing text

Description automatically generated

Figure 16. VHDL code CPU controller pt 3

A picture containing timeline

Description automatically generated

Figure 17. VHDL code CPU controller pt 4

A picture containing table

Description automatically generated

Figure 18. VHDL code CPU controller pt 5

Text

Description automatically generated with low confidence

Figure 19. VHDL code CPU controller pt 6.

Text, whiteboard

Description automatically generated

Figure 20. VHDL Code Single Cycle CPU pt1.

Text

Description automatically generatedText

Description automatically generated

Figure 21. VHDL Code Single Cycle CPU pt2.

Text

Description automatically generated

Figure 22. VHDL code package pt1

Text

Description automatically generated

Figure 23. VHDL code package pt2

Text

Description automatically generated

Figure 24. VHDL code package pt3

Graphical user interface, application

Description automatically generated

Figure 25. Successful compilation report.

# Block Diagram & Simulation:

In this section I will show all simulation results.

A picture containing timeline

Description automatically generated

Figure 26. Block diagram for adder

## Adder:

Graphical user interface

Description automatically generated with medium confidence

Figure 27. Adder simulation.

This is the adder simulation. As you can see in the simulation, adding 11 and 1 give us 12. In terms of two 32 bit inputs, we get 0x00000011 + 0x00000001 = 0x00000012. Our second example is 0x00000003 + 0x00000002 = 0x00000005. Address Logic and Program counter uses adder as a component.

## 2:1 Mux:

A picture containing graphical user interface

Description automatically generated

Figure 28. 32 bit 2:1 mux bsf.

Graphical user interface

Description automatically generated

Figure 29. 32bit mux in modelsim simulation.

Above is bsf and simulation for 2:1 mux. This will help the data being selected during instruction execution time. The mux will help guarantee data chose is appropriate for the instructions execution. The value switches when the selector is changing from 1 to 0 or vise versa.

## Program counter:

A picture containing application

Description automatically generated

Figure 30. program counter bsf

A screenshot of a computer

Description automatically generated

Figure 31. program counter simulation.

This will update the value of program counter which leads to the next instruction. Program counter value will output during rising edge of clock. Output is then sent to instruction memory and adder before returning to program counter as an input.

## 5bit Mux:

A picture containing text

Description automatically generated

Figure 32. Block diagram for 5bit Mux

A screenshot of a computer

Description automatically generated

Figure 33. 5bit Mux simulation.

In these two above figures, we see the block diagram and the simulation. Our input in this case is 5-bit. I put selector set to clock so that we can see the output change. If Selector is 1, signal A is displayed and if selector is 0, signal b is displayed. Register destination inside of CPU control is handled by this component.

## Next Address Logic:

Timeline

Description automatically generated

Figure 34. next address bsf.

A screenshot of a computer

Description automatically generated

Figure 35. Next address simulation

If condition is 0, Program counter + 4. If condition is 1, PC +4 + Extended Immediate “00”

## Instruction Register:

Graphical user interface, application

Description automatically generated with medium confidence

Figure 36. Instruction register bsf.

A screenshot of a computer

Description automatically generated

Figure 37. instruction register simulation.

This is our instruction register. This will take instructions and decode it for ALU and assign RS,RT, and RD.

## Instruction Memory:

Timeline

Description automatically generated with medium confidence

Figure 38. instruction memory bsf.

Graphical user interface

Description automatically generated

Figure 39. instruction memory simulation.

This is the Instruction Memory's file. This is where we put the MARS simulator code for the dot product. As a result, the values displayed here correspond to the MIPS assembly code for computing the dot product, which is included at the end of this file. Because the Instruction Memory can't go higher than 32 bits, I'll comment this part out until I've added more operations for testing. Our typical ADD, SUB, and MUL operations are commented out in these parts.

## Register File:

Graphical user interface

Description automatically generated with low confidence

Figure 40. register file bsf.

The register file is a 3-Port RAM file. With a total capacity of 32 32-bit data, it has one input port and two output ports. The 5-bit addresses Ra,Rb are used to read the Register values using busA and busB, whereas Rw is used to write into the Register using busW. This type of RAM can read and write two data sets at the same time. The ALU and memory to register writing will be done on the buses. In this situation, all registers are set to 0.

## Data Memory:

A picture containing graphical user interface

Description automatically generated

Figure 41. Data memory bsf

Graphical user interface

Description automatically generated

Figure 42. Data memory simulation

As demonstrated in the waveform above, the address is the same for both read and write access. We assert the write enable when we want to write to a specific address in Data Memory. The memory reads and prints the value based on the 4-bit address specified. The array is used to construct the data memory instead of the LPM Module.

## ALU:

The ALU unit has 10 operations. The table below will show the opcode for the operations that will be used by the CPU Controller.

Table

Description automatically generated

Graphical user interface, application

Description automatically generated

Figure 43. ALU bsf.

Graphical user interface

Description automatically generated

Figure 44. alu simulation.

The figures above show the alu unit operations. We don’t obtain overflow because I did not use values that give us an overflow. My ALU unit calculates the OVERFLOW for ADD and SUB instructions, including the unsigned operation. To distinguish between unsigned and signed, an Overflow Unit is built to output the resulting overflow, which is reliant on the opcode and function from the instruction.

## Sign Extender:

A picture containing application

Description automatically generated

Figure 45. sign extender bsf

Graphical user interface, treemap chart

Description automatically generated

Figure 46. Sign extender simulation.

Sign extender allows us to stretch a 16bit input to a 32-bit output. The selector determines how this works. The CPU controller will decide if the number must be extended to zero or extended in sign direction.

## Overflow:

There is no overflow bsf or simulation since it must be used injunction with other components. However, this file will check for overflow when doing addition.

## CPU Controller:

The CPU Controller is the brain of a single-cycle CPU. It determined the proper data path for the instructions being run. Multiplexers, Extenders, ALU Controller, Memory, and Register writes are all overseen by this.

Table

Description automatically generated

In the table above, the values required for the Register Write, Memory Write, PCSrc, ALUSrc, Memory To Register, Register Destination, and Extension operations of the 18 MIPS instruction are listed. The table was used to design the CPU Controller unit.

## Single Cycle CPU:

Graphical user interface

Description automatically generated

Figure . Single Cycle CPU waveform.

This simulation above shows given opcodes and functions and the output that is given. For opcode 000000, it is an R-type instruction so it relies on the function. However, if we test opcodes 00100, 001100, and 000100, they do not rely on the function since they are I-type instruction based. One opcode that I did not show was the j-type instruction, but the opcode is 00010.

## Component Package:

Included in the component package are the following components:

* 32-bit adder
* 32bit Mux
* Program Counter
* 5bit Mux
* Next Address Logic (NAL)
* Instruction register
* Instruction memory
* Register File
* Data memory
* ALU
* Sign Extender
* Overflow
* CPU Controller

# Single Cycle CPU:

In this section, I will demonstrate the CPU and its required instructions.

## ADDI, ORI, ADDIU, NOR, LW Instructions:

Text

Description automatically generated

Figure . Updated code for instruction memory for instructions.

Text

Description automatically generated

Figure . Updated data memory for instructions.

Graphical user interface

Description automatically generated with medium confidence

Figure . Modelsim waveform for ADDI, ORI, ADDIU, NOR, LW Instruction

In this simulation we see the PC pointing to the next address and hold desired instruction. You can see the ADDI instruction with the use of the IMM16. We can also see the ORI with the IMM16 . We load values from our data memory, into the register. ADDIU is also shown as it does addition and won’t trigger overflow flag.

## LW, ADD, ADU, SW Instructions:

Text

Description automatically generated

Figure . Updated instruction memory code for instructions.

Text

Description automatically generated

Figure . Updated data memory code for instructions.

Timeline

Description automatically generated

Figure . Modelsim simulation for LW, ADD, ADDU, SW Instruction.

The waveform above shows that for our first clock cycle, nothing is changing since we can not perform anything. The PC will hold the next address due to our components. The second instruction s then given when the second clock cycle begins. For me that is address 0x8C290000. This loads the value of data memory which if you recall from the edited VHDL code above, it is 0x7FFFFFF since it was initialized data. The PC now moves to the following address 0x00000004. Here, $9 is loaded onto the register in the next clock cycle and points to the next address. This process continue to repeat. The ADDU operation was used for the next instruction to obtain the same result as ADD except for triggering the overflow flag.

## BEQ, BNE, J, SUB:

A screen shot of a computer

Description automatically generated with low confidence

Figure . Modelsim simulation BEQ, BNE, J, SUB instructions

# Ultimate Test:

A picture containing text

Description automatically generated

Figure .instruction memory updated code

Text

Description automatically generated

Figure . data memory updated code

Graphical user interface

Description automatically generated

Figure . Modelsim simulation example

# Mips Summation:

Table

Description automatically generated

Figure . Summation code .asm

Graphical user interface, application, table

Description automatically generated

Figure . text segment

Graphical user interface, application, table

Description automatically generated

Figure . data segment

# Conclusion:

I learned a lot from this project. Using all components that we have previous used has allowed me to create this Single Cycle CPU and it taught me to continue to reuse code experience that we had used. I got a better understanding on how to use advanced ideas in Quartus and I was able to complete the given task in the final lab.