Fall 2013

|  |
| --- |
| ECGR 4146 | Gustavo Cruz, Brian Nguyen, Madison Traynham |

|  |  |
| --- | --- |
| Final Project | RISC PROCESSOR |

Table of Contents

[Objective 2](#_Toc373881064)

[Project Requirements 2](#_Toc373881065)

[Background 2](#_Toc373881066)

[Instruction Set 3](#_Toc373881067)

[Stages 4](#_Toc373881068)

[Fetch Stage 4](#_Toc373881069)

[Decode Stage 6](#_Toc373881070)

[Execute Stage 8](#_Toc373881071)

[Writeback 10](#_Toc373881072)

[Simulations 11](#_Toc373881073)

[Fetch 11](#_Toc373881074)

[Program Counter and Branch 11](#_Toc373881075)

[Instruction Memory 11](#_Toc373881076)

[Decode 11](#_Toc373881077)

[Decoder 11](#_Toc373881078)

[Execute 12](#_Toc373881079)

[Register Bank 12](#_Toc373881080)

[ALU 12](#_Toc373881081)

[Writeback 12](#_Toc373881082)

[Data Memory 12](#_Toc373881083)

[Pipelining 13](#_Toc373881084)

[Conclusion 14](#_Toc373881085)

[Works Cited 15](#_Toc373881086)

# Objective

The final design project for ECGR 4146 Introduction to VHDL was to design, implement and test a 16 bit RISC (Reduced Instruction Set Computer) Processor, with four stage pipelining and interrupt handling. This project was executed in VHDL using Mentor Graphics Modelsim SE 10.1b design software.

# Project Requirements

Following the project guidelines given in the RISC Processor Final Project handout, the final design must include the following:

* 16-bit Instruction Set
* 4 stages of pipelined registers
* 4 external interrupts
* Non preemptive interrupt handler
* 16x8-bit register
* 8-bit wide data memory
* 8-bit wide address bus

# Background

A RISC processor is based on computer architecture that uses a set of small and simple instructions that can be executed quickly and efficiently by using single cycles and pipelined stages to process instructions. Prior to the design of RISC the instruction sets of many computers were much more complex and instead of low level operations. The most common form of RISC processor is MIPS, or Microprocessor without Interlocked Pipeline Stages. The instruction set in use during this project focuses mainly on four types of instructions the R-type, Branch, Immediate, and EM-type.

## Instruction Set

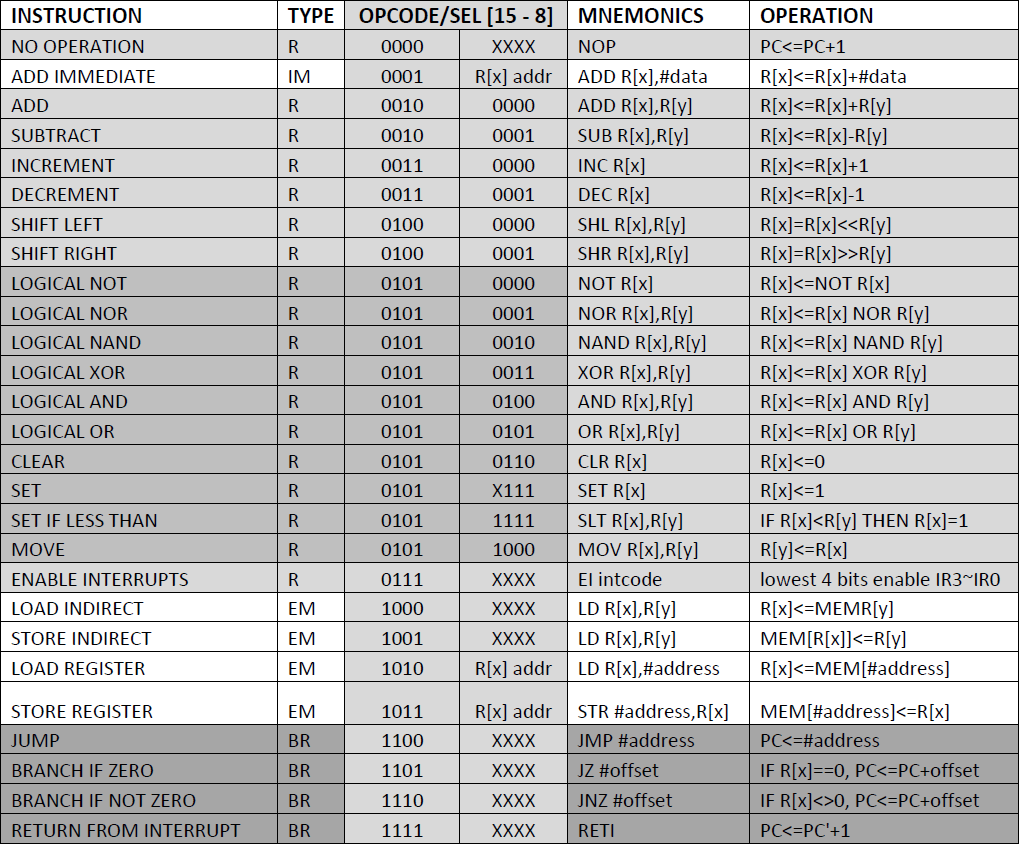


Figure 1. Instruction Set for RISC processor

During the design of the processor the above instruction set was followed. This set is the basis for the entire processor design and operation. In the implementation of the RTL design process there are a few basic components that are linked together. The controller section of the processor is located mostly inside of the decoder while the datapath components make up the rest of the design.

# Stages

During the construction of the processor we assumed that there must be 4 stages. The components created in the datapath and controller was wired together in these stages to create each stage. Once each stage is created and verified working the stages can be connected together through pipelined registers.

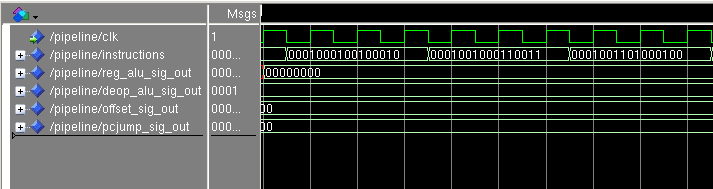
## Fetch Stage



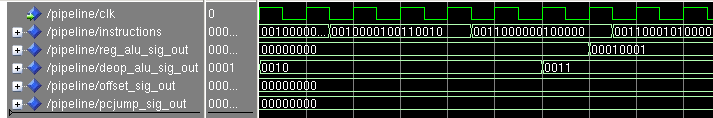
Figure 2. Fetch Stage Diagram using a Program Counter with integrated Branch and Instruction Memory

In the fetch stage the components used included the pc counter, branch operation and instruction memory. These components were wired together in a top-level file named “Fetch.VHD” which follows the wiring schematic outlined in the overview diagram.

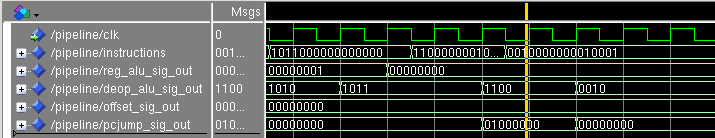
Changing Instructions out indicating increasing PC counter.



For these three add immediate instructions no input the signals



Instructions change to add operations and input signal in to FETCH stage receive values though not used.



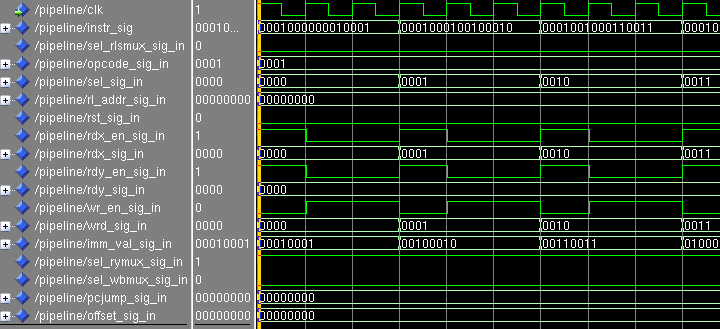
Instructions reach a jump instruction.

Jump location being sent into FETCH stage

## Decode Stage



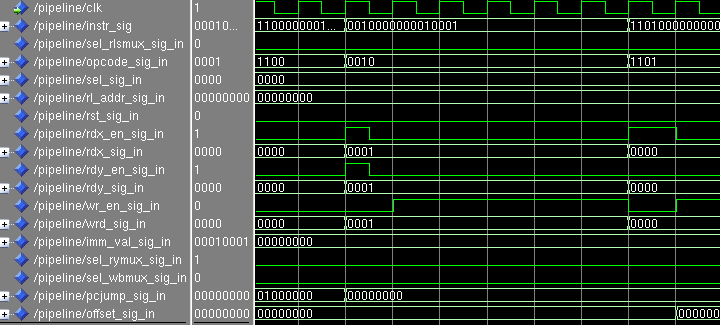
The decode stage follows the same procedure of wiring the components to a top level file named “Decoder.vhd”, the only file included in this file is just the decoder as it handles all of the inputs coming from the first pipeline register and outputs multiple signals to control the operations pertaining to the instruction.



Here is the DECODER stage taking in instructions coming from the FETCH stage and decoding them to then send out the appropriate signals values to accomplish the operation.

This instruction is the add immediate instruction, the decoder breaks it down and sends out the appropriate opcode, sel, immediate value and register bank information to complete the operation.

This instruction is the jump instruction; the decoder breaks it down and sends out the appropriate opcode, sel, and jump value to complete the operation. The jump operation sends the instruction signal to the add immediate instructions and the decoder sends out the appropriate signal values then.

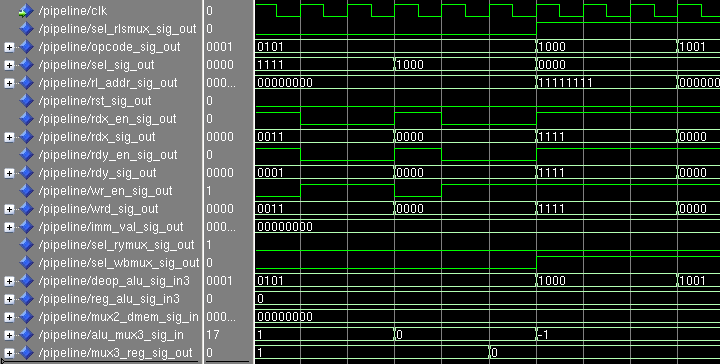
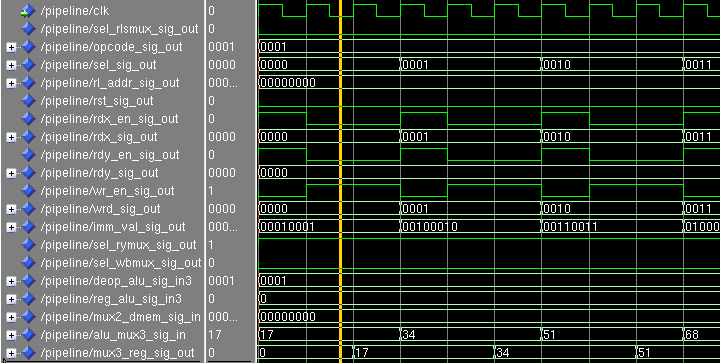


## Execute Stage



Figure 3. Execute Stage data flow with signals and components labeled.

The execute stage, located in the “Execute.VHD” file, is comprised of a register bank which stores values to be used by desired operations. These values are then taken and passed through the ALU according to the opcode and select lines decoded in the previous stage. The results are passed to the next location using the signals outlined in the figure.



The execute takes in the appropriate signal values from the decoder and does the appropriate operations and outputs them to the next pipeline register.

## Writeback

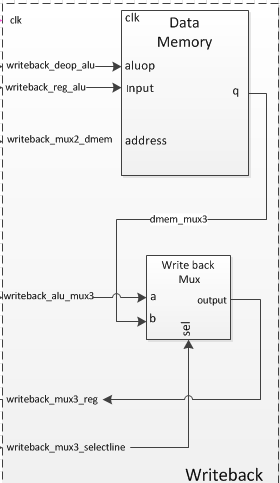
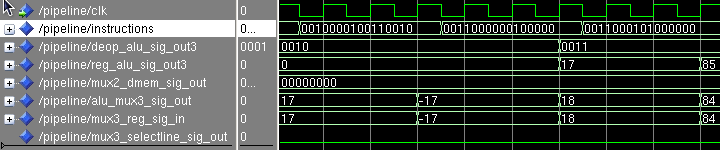


Figure 4. Write back stage diagram with Data Memory, Write back Mux, and associated signals.

Finally, as with the other stages of the processor the write back stage was formed using port mapping and a top level file named “Writeback.VHD”. The write back stage uses the data memory to store related information and a mux to determine how the data will be written back into the register.



At the WRITEBACK stage its takes in the input coming out of the pipeline register and does the appropriate operations depending on the opcode coming out of the DECODER stage.

# Simulations

## Fetch

### Program Counter and Branch

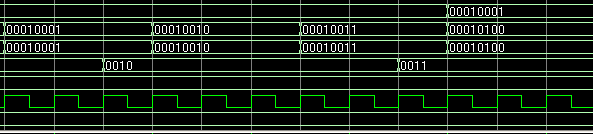
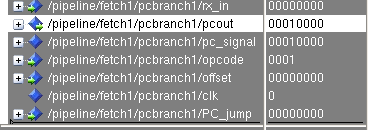


Figure 5. The Program Counter is incremented by 1 every time the processor completes a cycle. This effectively moves the program forward and allows for the next instruction to be run.

### Instruction Memory



Figure 6. The instruction memory holds the values of the instructions to be processed. This is where the program to be run is stored. In order to run the program a value must be filled into the register to perform an operation on. The first 16 instructions in the memory are filling up the register bank by using the add immediate instruction. Afterword’s, the required instructions from the instruction set are added to use the previously loaded values.

## Decode

### Decoder

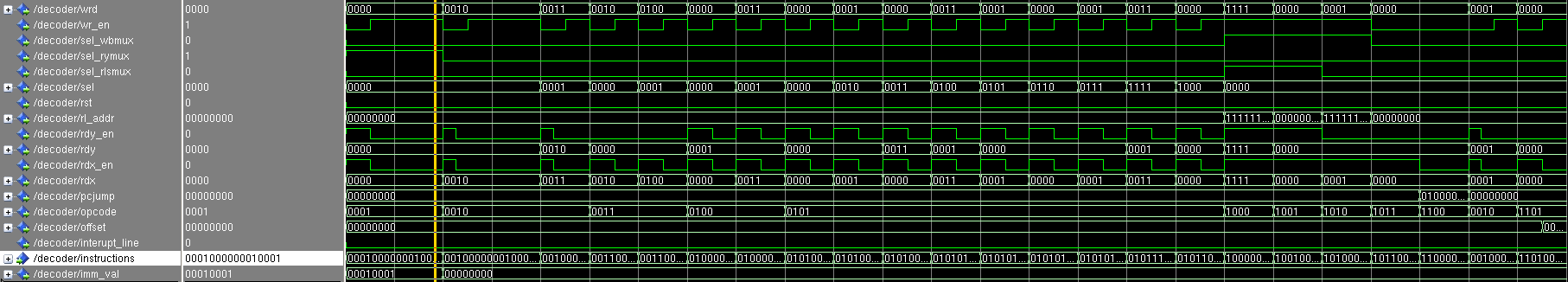


Figure 7. The decoder is bringing in signals from the instruction memory and breaking it up into its individual components. It uses the opcode and select line to locate the desired operation for the ALU while pointing the location for each instruction at RX and RY data locations in the register.

## Execute

### Register Bank

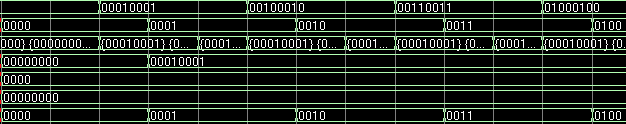


Figure 8. The register bank is where all of the data to be processes is stored. The register is always writing the data out to the connected lines however the read enable is only turned on when necessary. During the write back stage a word can be written back into the register at a specified location.

### ALU

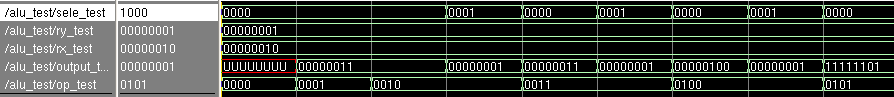


Figure 9. ALU Test bench to confirm that based on the opcode and select lines the proper operation will be performed.

## Writeback

### Data Memory

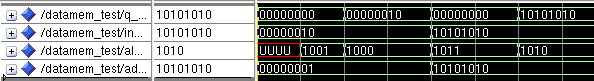


Figure 10. Results of Data memory test with testing of opcodes 1001 and 1011 for store word and 1000 and 1010 for load instructions.

For most of the stages there was at least one mux used. Due to the simplicity of the mux and its operation a basic mux test bench code will be used to show the functionality of the component. The location of each mux and its connected signals can be seen in the data flow diagram for each stage.



Figure 11. When the select line is at 0 the input from A will be passed through the mux to the next component. If the select line is changed to one the new output of the mux will be from input B.

# Pipelining

After successful verification of all components in the different stages pipeline registers can be inserted to achieve an increased throughput of the system. A pipelined processor increases the efficiency of the given resources and allows for more data to be processed once the initial execution time is overcome. The pipelined registers are placed between each stage and carry in the outputs of the previous stage and use them as inputs to the next, this implementation allows for multiple instructions to be inside of the processor at once. These operations are all delayed by a specified amount of time to allow for the writeback operation to occur and the next values to pass through accordingly. In the case of our pipeline implementation it was found to have a three second delay in each pipeline register to allow each instruction to pass through with enough time for execution.

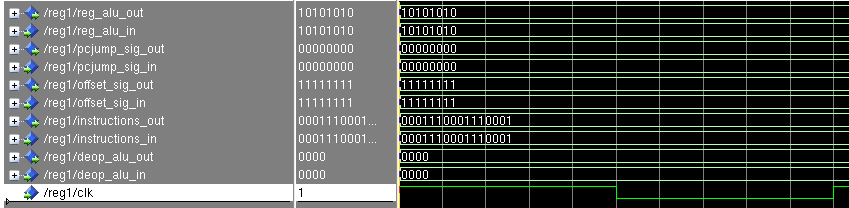


Figure 12. Pipeline register placed between the instruction fetch and instruction decode stages. The placement of this register brings in values from the fetch stage and passes to the input of the decode stage. The initial values as shown in the figure pass through the register immediately and then are delayed by one clock cycle after the change on the input in the remaining cycles.

Similarly the values for the Instruction decode/Execute pipeline stage can be seen below.

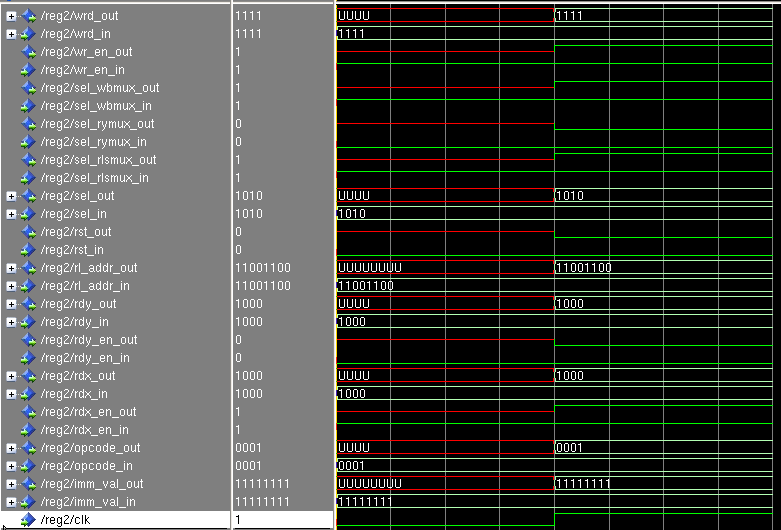


Figure 13. The secondary pipeline register stage using the clock to separate the output form the decode stage into the input of the execute stage. This operation is based on the rising clock edge and the values can be seen propagating throughout the register as expected.

Lastly, the third pipeline register was placed between the Execute and Writeback stages.

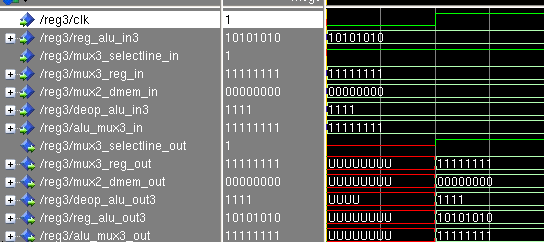


Figure 14. Pipeline register in between the Execute and Write back stages. This data is delayed by the register and then passed to the corresponding ports in the write back stage.

The sometimes common fourth pipeline register, in between the writeback stage and the instruction fetch stage was not assumed for our design. During implementation of the datapath and controller it was found to be unnecessary and ultimately caused catastrophic timing errors implementation was attempted. All assumptions, ports, and signal names can be viewed in the provided diagrams as well as in the matching VHDL code.

# Conclusion

The project began with the given instruction set for a RISC processor and expected components and functionality. Armed with this information the group used the given data flow diagram in the project outline as a starting point. After research and failed attempts at implementing the supplied dataflow a custom processor flow was created. Each component was built and thoroughly tested before implementation in the stages and eventually the pipeline. This method of building verified the working of the processor and ensured there would be no unforeseen problems. Overall the building and testing process took much longer than allotted for and the due dates created a bottleneck for the completion of the project. In result of this issue we were unable to fully implement the interrupt handler and scratch pad. If time allowed the desired final product would include not only a fully working interrupt handler and data hazard management system, but also an attempt to increase the throughput of the existing system would be made.

# Works Cited

Front Cover picture

<http://www.electronic-engineering.ch/study/silverbird/silverbird_chip.jpg>

or <http://www.electronic-engineering.ch/study/silverbird/silverbird.html>

RISC Architecture

<http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/risc/whatis/index.html>