Single Cycle CPU Design

Based on MIPS Instruction Set

**Introduction:**

MIPS stands for “Microprocessor without Interlocked Pipeline Stages” and is a very popular architecture in the embedded processor market that was first introduced in 1985. Although this architecture was first designed for general purpose computing as time went on it was instead used in embedded systems such as microcontrollers, wireless routers, LTE modems and automotive electronics. In this project we will be designing and implementing a similar type of single cycle CPU that is based on the instruction set of the MIPS architecture. By the end of the project we will be able to develop our own design of a CPU virtually using Verilog and then eventually emulate the CPU on the DE10-Lite board using its instruction set to control the hardware.

**Design:**

In order to complete the design there will be three different types of instructions that will need to be implemented; these are the A-format, B-format and L-format instructions. First we have the A-format instructions,

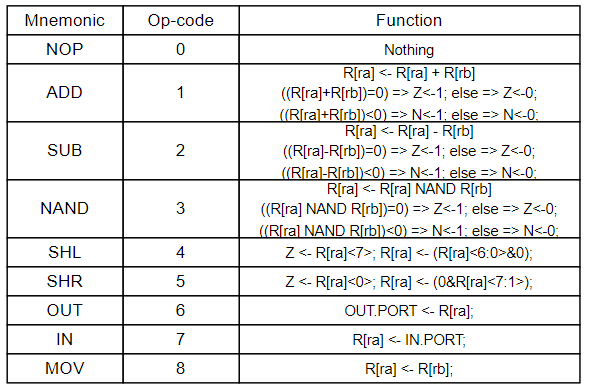


Fig 1: A-format Instructions.

These instructions are used for the arithmetic operations such as addition and subtraction as well as logical operations such as NAND, SHL and SHR, on top of that there is also data management for moving data in, out and between registers such as OUT, IN and MOV. All instructions are two bytes in the A-format only the first byte is used and is split up as follows,

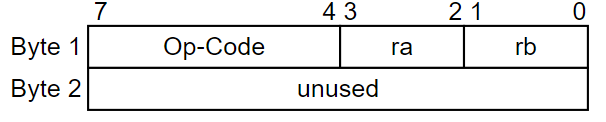


Fig 2: A-Format Bit Decode.

where the first two bits are used for the target register rb, the next two bits are used for the target register ra and the last 4 bits are used for the Op-code.

Next we have the B-format instructions,

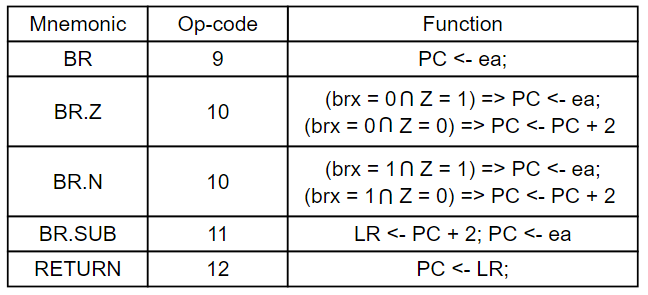


Fig 3: B-Format Instructions.

these instructions are used for various types of branching. These instructions are broken down as follows,

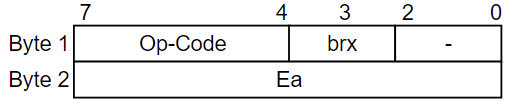


Fig 4: B-Format Bit Decode.

Where the first two bits are unused, the next two bits are used to distinguish between BR.Z/BR.N as they have the same opcode, and the last four bits of byte one are the op-code of the instruction, finally we have the last byte which is used for the location of the branch. When using BR the user specifies the location of the instruction that they would like to branch to, however, when using BR.SUB instead of just running the target instruction a special load register will store the location of the next instruction before branching. Then after the branch instruction is run the program will RETURN to the original location by using the location stored in the load register.

Finally we have the L-format instructions,

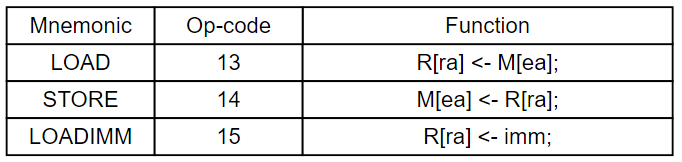


Fig 5: L-format Instructions.

These instructions are used for controlling the data stored in the data memory, first we have LOAD which will take data from a specific location in the data memory and move it into a target register in the register file. Next we have the STORE instruction which does the opposite and takes a value stored in one of the registers and stores it in a target location in the data memory. Finally we have LOADIMM where the user can specify a value and store it directly into a target register. Each of these instructions are broken down as follows,

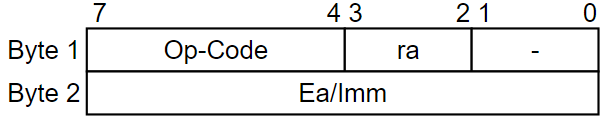


Fig 6: L-Format Bit Decode.

where the first two bits are unused, the second two bits are the target register, the last four bits are the opcode and the last byte is used for either the immediate value being stored or the location of the data being accessed in the data memory.

In order to develop this processor we first build the design of the CPU as seen below,

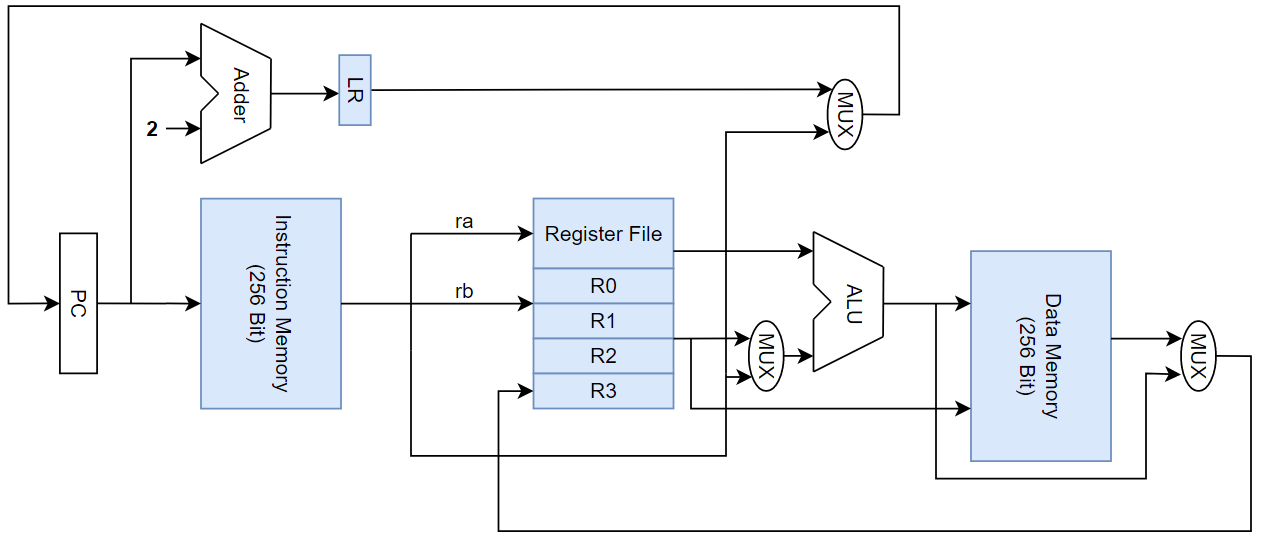


Fig 7: CPU Design.

In this design we have four memory blocks. First the instruction memory which is 256 bits is byte addressable and is used for storing and decoding the incoming instructions. Next we have the register file which is composed of four single byte general purpose registers and is used for quick data access. After that we have the data memory which is also 256 bits in size and byte addressable, this is the main memory of the system and is used for storing data. Finally we have the load register LR and is used to store the value of the next instruction and can be accessed in the case of a subroutine call in order to properly return to the correct instruction after performing the subroutine.

On top of the memory we have the ALU which is responsible for performing the various arithmetic and logical operations, the program counter which tells the program which instruction to run, the adder which will increment the PC by two each time an instruction is run and finally multiple MUX which help the flow of the data through the control lines from one module to another.

Next we will also take a look at the control lines that are used in each of the different instruction formats starting with the A-format instructions,

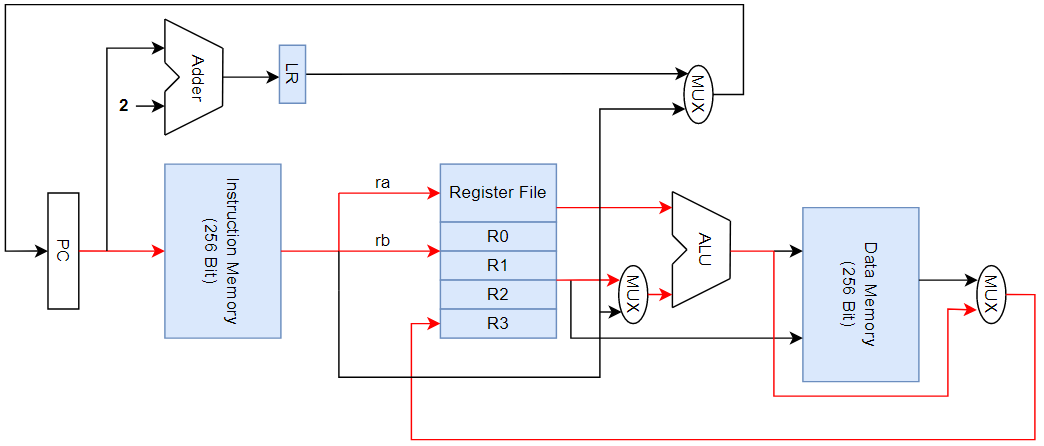


Fig 8: A-Format Datapath.

In this case the instruction is passed through the instruction memory where it is decoded, then the two target registers ra and rb are selected, the data from the target registers are then passed to the ALU. The controller for the program will tell the ALU which operation to perform based on the decoded instruction from earlier. After the operation is performed the output of the ALU is sent back to the register file for write back to ra if the write back flag was selected by the controller.

Next we have the B-Format instructions,

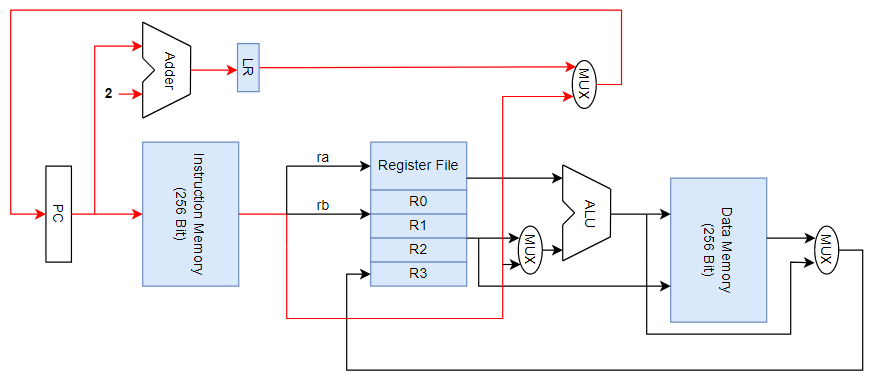


Fig 9: B-Format Datapath.

In this case when we branch the target instruction location is passed in the second byte of the instruction to the PC, after that the instruction corresponding to the value passed to the program counter is run instead of the next instruction. Then in the case that it is a branch with subroutine before branching the location of the next instruction is saved in the load register, after the target branch instruction is run the value in the load register is passed back to the program counter and the instructions then continue running as they were before the subroutine had started.

Finally we look at the datapath for the last instruction format L-Format,

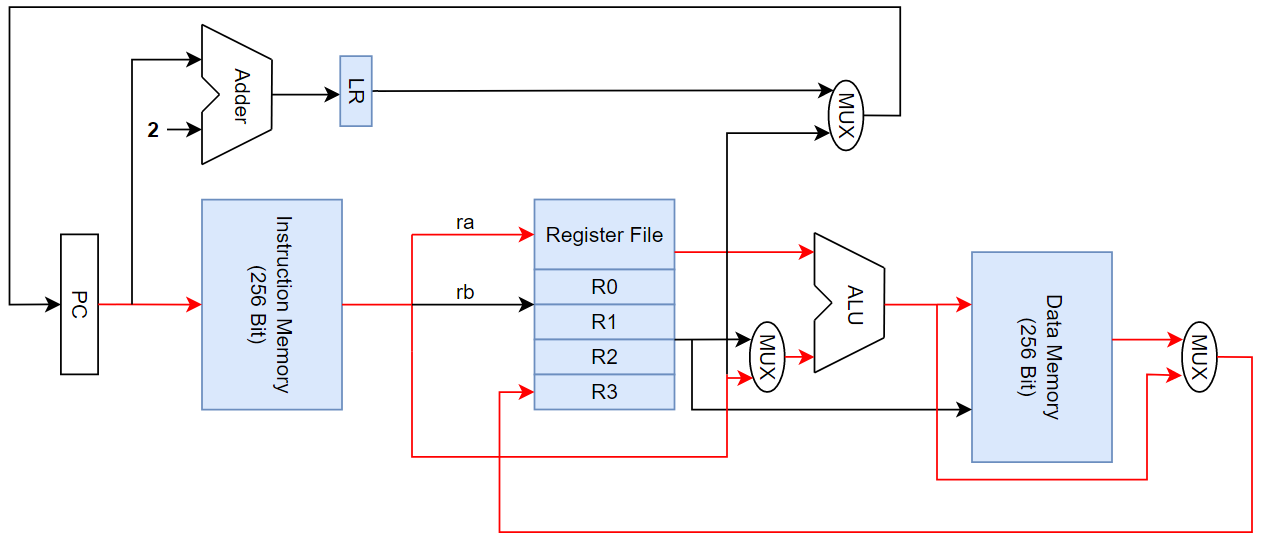


Fig 10: L-Format Datapath.

Here we pass the target register ra which will either be accessed for the data in it or used for write back, then we pass the memory location to the data memory which is then either used to store the data passed from ra or the data at the location in the memory is accessed and is then passed back to the register file for write back into the target register ra. Lastly we can also pass an immediate value into the CPU along with a target register where the value will be directly stored into the target register. In both of the A-Format and L-Format instructions the program counter is also incremented by two.

**Debug:**

After completing the design of the CPU it was then time to create the processor using Verilog. There were many struggles during the development process however a few of them were more important than others. First of all when it comes to the data blocks for the instruction memory and the data memory there were a few different methods for design before coming to the final iteration that functioned correctly. Creating a block of data that was 256 bits long was easy however when it comes to making it byte addressable there were possible issues with data overflow into neighboring bytes of a data as well as accessing a specific byte of data was more tedious. Instead I found it was better to take the entire block of data and partition it into 32 one byte blocks of data making them much easier to access for data storage or retrieval.

Another issue that I ran into during the development process was troubleshooting issues as it was initially very difficult to track the flow of data as well as the final states associated with each value. Initially I was creating temporary outputs in the top module that would have various values from other modules passed to them so when the program was run in modelsim I could check the value to make sure that it was correct. This was ok when the program was still smaller and had a lot less movement of data however when more modules were added having them all also outputting to the top module and including them in the test bench was too time consuming and another method needed to be used. After doing some research regarding modelsim I found that you could instead include all of the other signals in the design with the input and the outputs of the top module, this would allow you to see the final states of each variable or block of data storage making debugging and troubleshooting far easier.

Finally the last major roadblock that I ran into was related to the passing of data between each of the modules. Originally when trying to instantiate another module to pass the input of a module to the output of another module I was instantiating the top module in a different module instead of the other way around. This led to improper instantiation and the data would then not be passed correctly instead I needed to instantiate the two modules I wanted to connect into the top module then use a wire in order to connect them and pass the data correctly. This was all fine until I needed to pass an output from one module to the input of my instruction memory module, at the time this was the first module I created therefore it was still my top module. Since it was the top module the inputs for it had to be passed to it through the test bench instead of from another module. In order to fix this I had to create a new top module dedicated to only the passing of data between modules and then move the code in the old top module that was associated with the instruction memory into a new module which would become the instruction memory module that I have now. After making this change the flow of data was functioning correctly for all modules including the instruction memory, it also made the code less confusing as all of the module connections were not also being done in the instruction memory.

**Conclusion:**

After developing this project there are a few major takeaways that I learned that I believe would greatly aid in the development of other computer architecture designs in the future. First of all is making sure that the original design is as good as possible before starting the development. It is very important to have a well built design as if you are constantly finding issues with the design during the development stages then it will take a lot more work at the time to have to go back and change the design as well as then refactor those changes into the current development. This can lead to needing to rewrite sections of the code that may already be built in the way they were originally designed, greatly slowing down the development process.

Another very important thing that I learned from this and would do differently if I were to do it again would be to plan out the implementation of the code using pseudo code in order to help stay true to the design. This would also help to limit any confusion during development as you have already created a kind of template for how it will be implemented. On top of that any changes that would then need to be made could be made first in the pseudo code to help plan how those changes will be done before finally adding them to the final design.

In conclusion I believe that this project has greatly improved my knowledge and understanding of computer architecture as well as Verilog in general. Before this I have only developed smaller designs using Verilog and that is a very different experience, there is not a necessity for the designing and planning before the implementation as the smaller designs do not have the same complexity. On top of improving my designing I also learned a lot practically about writing code in Verilog especially when it comes to the passing and control of data in the system. Finally with these things I have learned there are certain practices that I would be sure to use in the future when creating another system like this such as better naming schemes as well as the use of pseudo code before jumping into the actual implementation of the code in Verilog.