# INF-2200-1 22H: Computer architecture and organization

# Assignment 2

Amund Strøm

UiT id: ast283@uit.no

GitHub users: ast283

GitHub Classroom: assignment-2-amund

October 17, 2022

## 1 Introduction

MIPS is one of many different computer architectures which tells how to implement a processor, but what sets MIPS apart is its simplicity and easy to learn architecture. This report details the implementation of a simplified version of a MIPS architecture. The assignment tasked us to create a pipelined version and support some basic instructions. My implementation supports the requested instructions, but it is not pipelined.

## 2 Methods

### 2.1 Design

The MIPS architecture consist of seven important elements. These are the Program counter, Instruction memory, Control unit, Register file, Arithmetic logic unit (ALU), ALU control unit and Data memory. In between these elements are smaller and less impact full elements, but are just as necessary to implement a working processor.

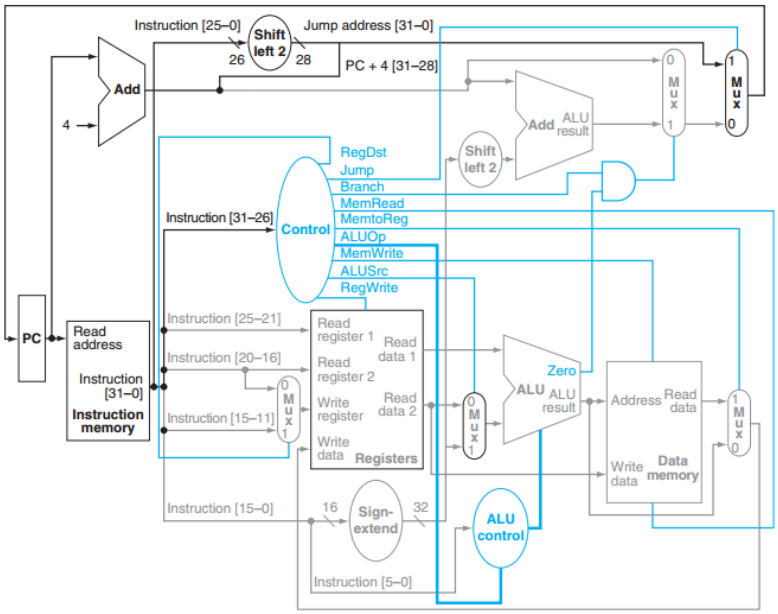


Figure 1: An illustration of the datapath of a MIPS architecture that is not pipelined.

Each of these elements have very a specific and simple purpose, but if we connect them all together we get a working processor. Each element have an input and an output, either in the form of a control signal or raw data. For example the program counter have one input and two outputs, both in the form of raw data. On the other hand you have the control unit, that have one input of raw data, and 9 control signal outputs. Control signals have the function to determine what kind of operation an element is supposed to do. Meaning the control signal choices how the raw data of an input should be manipulated inside an element.

Knowing this information when implementing the processor, we simply implement one single element at a time, and test if it works by giving it inputs where we know what output it should return. After every element is tested, connect them according to the illustration. Something that is important to keep in mind when connecting the elements is to know beforehand what type the inputs are, meaning that if the inputs come in the form of a string, but you was expecting an integer, you program will fail. An easy way to handle this problem is to always send the same type of outputs, regardless of what type the next element is expecting.

As mentioned in the introduction, the assignment tasked us to complete two different tasks. The first being to support some, but not all, instructions of the MIPS architecture. This means that the processor can execute operations such as adding two numbers, jump to next memory address, load and store words, the list goes on. The point is to implement the basic instructions, since implementing them all is time consuming and not difficult when you have already implemented the basics.
The second task was to implement a piplined version of the datapath, which means that the second an element is finished with an instruction, it will immediately start on the next instruction, rather than to wait for all the elements to finish. A single cycle datapath does exactly that, it will wait for all the elements to complete, before starting on the next instruction, making it extremely slow compared to the first.

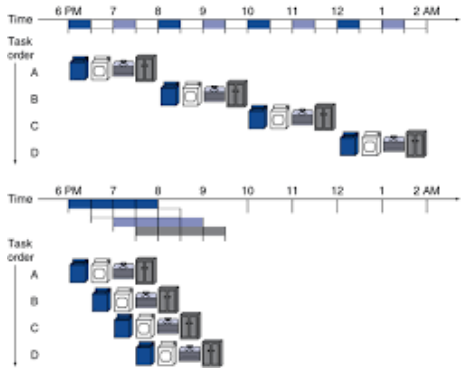


Figure 2: An analogy between the difference of a single cycle datapath and a pipelined datapath. The first being a single cycled and the second being a pipelined datapath.

## 2.2 Implementation

My implementation matches the design quite well. It is a single cycled datapath because I did not have time to implement a pipelined version. It supports all the requested instructions except one. To keep things tidy, every elements output is represented as an integer. Finally every element work as intended with no known errors.

Just because the implementation works, does not mean that there was no implications during the creation of the program, and there are some choices that I have done that may not be that obvious. Figure 1 is essentially the solution to this assignment, if you implement all the elements that are represented in the illustration, you can execute most of the instructions. But there are some exceptions (that I know of), these being the jump and load upper immediate ("lui") instructions. To execute these instructions you need new elements that that manipulate data that no other elements do. This does not mean that the elements are complicated, just unique.

The jump instructions is the most obvious, since you can see on the illustration that there is a cross section of two different memory addresses that need to be connected. Therefore we create an element that takes two inputs of raw data which will be memory addresses, remove the unnecessary parts of the inputs, connect the remaining important parts, and set as output.

The load upper immediate instruction loads any value onto any register, but the value that is to be loaded is left shifted 16bits. Therefore we create an element that takes one input of raw data which will be an 16bit value, and shift it 16times to the left, this will create an 32bit value that will be the output. Since this value is to be instantly loaded onto an register, there must also be another multiplexer (mux) that determines if the information that is to be loaded onto the register is either from an "lui" instruction or from the multiplexer that chooses between the ALU and data memory.

Mentioned earlier, my implementation supports all the requested instructions except one, that is the not or (nor) instruction. The reasoning for this is that the "add immediate" and "not or" instruction uses the same opcode sent to the control unit, which means that there is no way of telling them apart when setting the different control signals. Since they don't use the same control signals I chose to ignore the "not or" instruction and not implement it.

### 2.3 Experiments

During the implementation of the program I tested each element individually. By giving the elements inputs where I knew what output to be expected. After connecting all the elements I ran every test that the pre-code provided, and check if the output was as expected.

## 3 Discussion

The main focus of the assignment was to get the simulator working and optimise it later if there was still time. As it turned out I did not have enough time to implement a pipelined version of the program. As mentioned in the design section, a pipelined version of an datapath is way more efficient than an single cycle datapath. Since it will execute the next instruction as soon as an element is finished with its current instruction. This of course comes with some implications that will prevent starting the next instruction in the next cycle, such as structure hazards, data hazards and control hazards.

### 3.1 Hazards

Structure hazards occurs when two or more instructions require the same element to complete its task at the same time. Situations like these happens when for example an write is made to the register file at stage 5 of an instruction, at the same time as an read is made on stage 2 of an instruction on the register file, accessing the same register as the write instruction, which value will be read? To handle this problem an approach could be to stall, meaning to add an "no operation" between these two instructions. Or to write onto the register on the HIGH of the clock cycle and read on the LOW, but this would be difficult to implement on an simulator such as this assignment.

Data hazards occurs when an instruction depends on data that needs to be completed by a previous instruction. Assume that an instruction writes onto an register, and the next instruction requires that data written to the register to preform its task. Since write back is on stage 5 and read register is on stage 2, there is no data to be read on the given register. To handle this hazard an approach could be to yet again stall and wait for the write back step to occur. This is easy to implement but will slow down the execution time of the program. Another approach called "Bypassing" uses the data right after it is computed, rather than waiting for it to be stored. This is harder to implement and requires extra connection between elements in the datapath, but is more efficient than stalling. But in some cases an combination between these two approaches must be implemented.

Control hazards occurs when a branch is taken or not, since instructions determine the flow of control and fetching the next instruction depends on the branch outcome. Assume that a program is supposed to take a branch, this decision is evaluated at stage 4, which means there are instructions inside the datapath being executed that are not supposed to be executed. There are essentially two different approaches to solve this, again we can stall the program and wait for the branch to finish, this is easy to implement but will slow down the execution time. The other approach would be to predict if the branch will be taken or not. This approach also have two different versions, a static branch prediction and a dynamic branch prediction. Static prediction treats every backwards branches as taken (such as for-loops) and forward branches as not taken (such as

if-statements). If the prediction is wrong, fill the pipeline with "no operations" and continue. Dynamic prediction on the other hand records the result of an branch and assumes the trend will continue. And if the prediction is wrong, update the trend for that branch and fill the pipeline with "no operations" and continue. Both of these version are faster than always stalling the program when an branch is encountered.

## 4 Conclusion

In this assignment I have implemented a program that simulates a single cycle datapath using the MIPS architecture, the simulator also supports the requested instructions. A single cycled datapath is executes much slower than an pipeleined version, but I did not have time to implement a pipelined version.

## References

[1] David A. Patterson, John L. Hennessy. Computer Organization and Design MIPS Edition The Hardware/Software Interface, 6th edition. Chapters 4.0-4.12.

[2] Lars Alio Bongo. The Processor.
from `https://docs.google.com/presentation/d/1Ykl4SJQD25TwkFOPQtgSd4IWbBgX2VzGDPoqFL6BNKc/edit#slide=id.p43`

[3] Dr A. P. Shanthi. Handling Control Hazards. 17.10.22.
from
`https://www.cs.umd.edu/~meesh/411/CA-online/chapter/handling-control-hazards/index.html`