06.10.2020 INF-2200

# Assignment 2 - MIPS

By Brede Eder Murberg & Joakim Kalstad Hjertholm

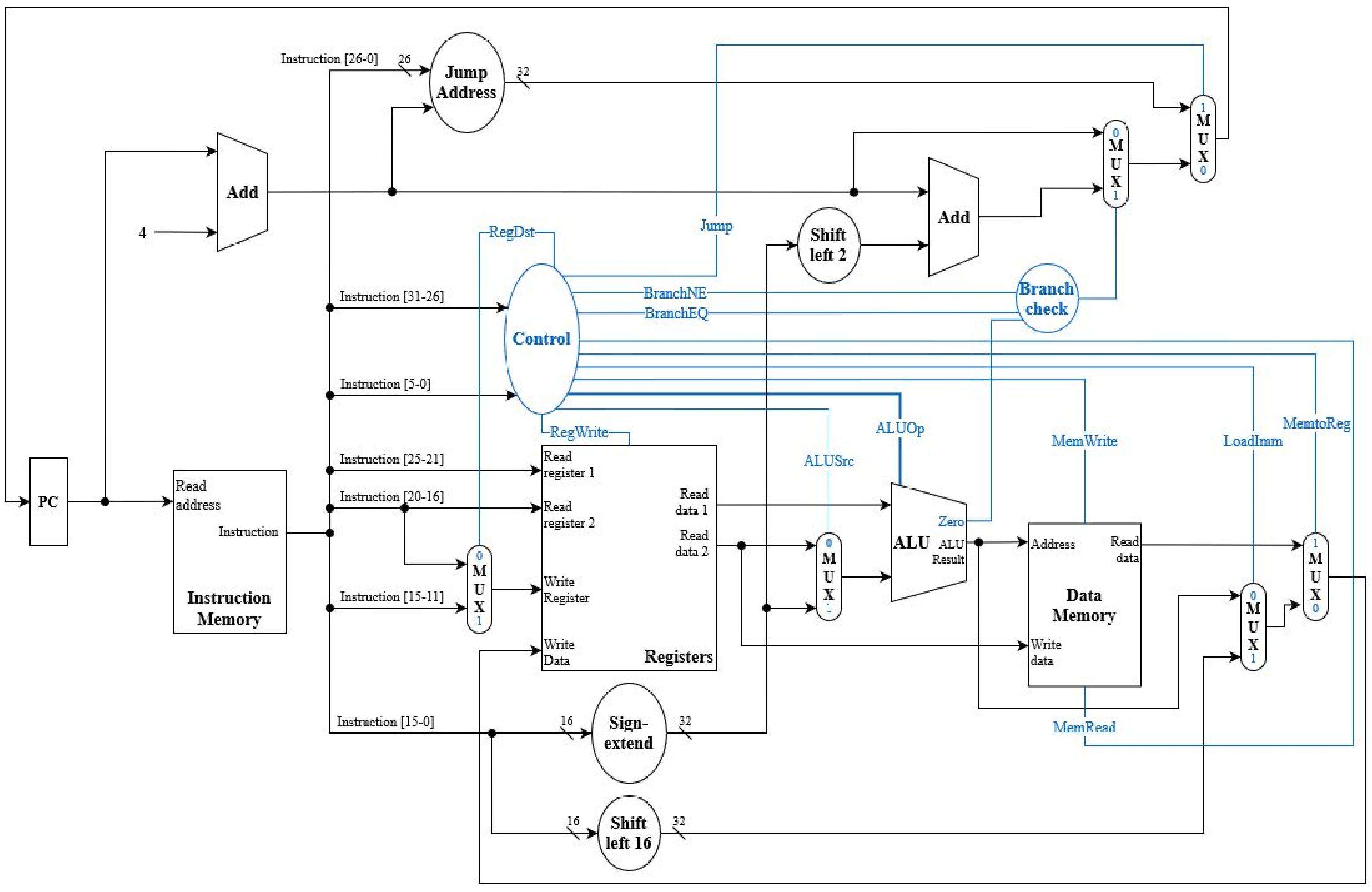
## Introduction

In this assignment we will simulate a subset of the MIPS architecture. The simulation will need to be able to read binary code and perform a set of instructions. It should also be able to simulate a pipelined datapath and controller.

## Design & Implementation

We have created two versions of the simulator. One is a single cycle implementation and the other is a pipeline implementation. We will now go over them separately.

### Single Cycle



The datapath consists of the following elements: PC, InstructionMemory, two adder units, control unit, registerFile, six multiplexores, ALU, two shiftLeft(2 and 16), signExtend, jumpAddress and dataMemory.

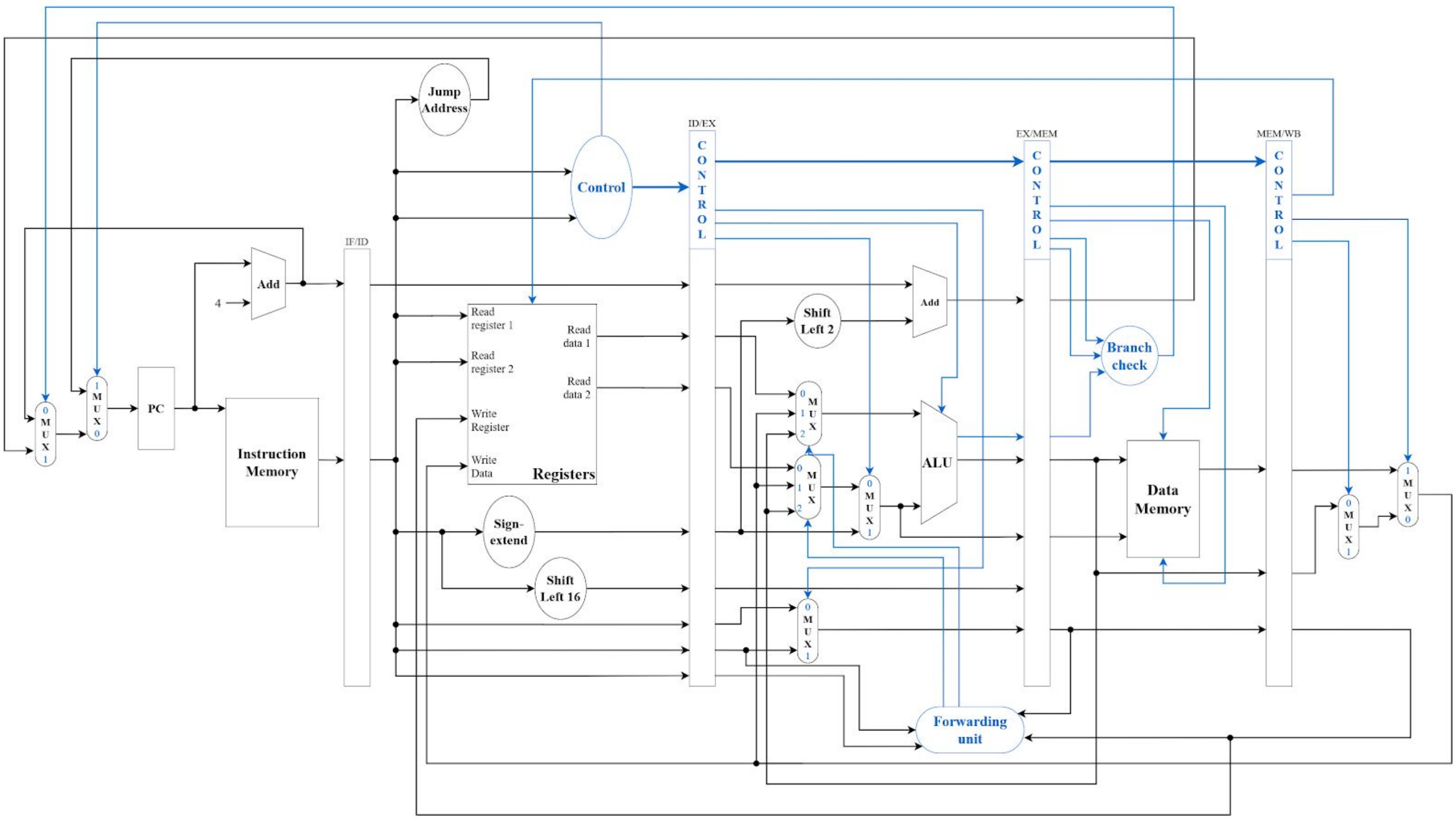
The datapath starts with the cpu element “program counter” (PC) outputting the address to the next instruction, and two cpu elements pick up this signal, “instruction memory” and “PCadder” (PC + 4). The instruction memory uses the address in the memory map to find the instruction mapped to the address. This instruction is then split up into different portions depending on the instruction type (R-Type, Add Immediate, Add Immediate Unsigned, LoadWord and StoreWord, Branch on Equal, Branch on not Equal, Jump or Load Upper Immediate). Then the instruction is split up and sent out in 7 output signals. Two of the seven output signals from instruction memory get sent to the control unit. These signals contain what type of instruction it is (fexp. R-Type), and what type of ALU function should be performed. Depending on the two signals, the control unit sends out control signals to almost all of the cpu elements. This makes them able to choose between input signals depending on the format and select what function or task it should perform..

Next we will enter the register file. It will get two register numbers from instruction memory, find the data that is stored on those registers and send the data out as two separate outputs. They will then enter the ALU, it will also take in a control signal which tells the ALU what type of function it should perform on the inputs (ADD, ADDU, SUB, SUBU, AND, OR, NOR and “set on less than” - SLT). The second input can be a sign extended address if the instruction is to LoadWord or StoreWord. Then the ALU performs its function and sends out the result. ALU also sends a control signal to the branch. This signal is only asserted if the result of the ALU is equal to zero. Data memory is the next to receive data, it will receive two inputs, one address from ALU and the other data from register file. Depending on its control signals, it will either read what’s in the memory on the address or it will choose to write the input data to the address location in the memory.

After completing datamemory we are going to run the register file again to write the result or data to the register. Before the output reaches the register we need to determine what data the register will write, this process involves two mux-es. First mux determines if it should get the ALU result or the address from instruction that has been shifted in leftShift by 16. The output is chosen by the control signal which tells us if the instruction wants a LUI (Load upper immediate). If not then the ALU result continues as input to the next mux. In the next mux it will send the output of the data memory if the instruction is a Load Word, otherwise it will forward the data from the previous mux. Then the register file receives the data and writes it to the register.

Now we have covered the lower half of the diagram that handles registers, memory and functions. The upper half calculates the next address for the next instruction. After the PC is added with the immediate four there are three elements that pick up the signal, these are jumpAddress, shiftLeft adder and the first mux. We start with the left shift adder which takes in sign extended address from instruction that has been left shifted by 2. Then add them together and output that to the mux. Then the mux forwards the normal PCadder to the next mux, except when it gets a branch control signal then the shift left adder output gets forwarded. The last mux chooses if it should use the jump address or not, and the jump address is calculated by using the lower 26 bits of the instruction and adding PC adder output (the Pcadder output was manipulated making the 6 bits the upper 6 bits of a 32 bit address). Then the jump address is the second input of the last mux, and the mux will forward the first signal to the PC, except when there's a jump control signal. With a jump control signal the output will be the jumpAddress. When the PC has received its new address and all the other elements are finished, the single cycle is complete.

## Pipeline



The pipelined version of the CPU uses the same elements as the single cycle version, but also has a few new ones. The most noticable is the four pipeline registers which split the CPU into five stages: Instruction Fetch, Instruction Decode, Execution, Memory access, and Write-back. These make pipelining possible by storing information between cycles. The different stages can operate on different instructions simultaneously, thus allowing five instructions to run at the same time. In our implementation however, it works a little differently. We still call on the CPU elements one at a time, but the order is altered. We go through the stages from right to left. Each stage starts by reading from a pipeline register, then it goes through the elements in the stage normally and ends by writing to the next pipeline register. Because we go from right to left, the pipeline registers will be read from before they are overwritten by the stage to the left of it.

To account for data hazards in the pipeline, we have implemented a forwarding unit. This cpu element checks if a calculated value is going to be put in a register which the following instructions are going to read from. If so, then the value is forwarded to where it needs to be.

To achieve this, we use two mux-es with three inputs and these decide whether to forward based on the control signals from the forwarding unit. Another way to handle data hazards is with stalling which stalls the next instruction until the value is written to the register. This is however much less efficient than forwarding which doesn't need to stall at all. Forwarding is on the other hand a more complex solution because of the added elements and connections.

Other notable changes are that the write register mux is moved to the EX stage and the output follows the pipeline to the WB stage. This is so that the write register and write data is sent to the register at the same time. Also, the branch- and jump-multiplexors are called on in the IF stage, thus deciding the next PC address at the end of a cycle.

Control hazard detection is something the pipeline implementation doesn't have. It was something we began to implement but had to cut due to a lack of time. The idea was that instruction memory would detect a jump, branch or LUI instruction and stall the pipeline until the instruction was done. Stalling is, like with data hazards, a slow solution. The faster way is to flush instructions. What that means is that instructions after f.ex. a branch are continued on like normal but if the branch is taken then the instructions and calculations after are erased or flushed. This is a more complex solution and in turn a more efficient solution.

## Testing

Each CPU element has a test to ensure that it does what it is expected to. All the tests are structured similarly. The element that is being tested is first connected to an input and output element. We can then decide what the input element sends to the component being tested and we can read from the output element and check if we got the expected result.

To test if the simulator is working, we can run it with the memory files from the precode. These are small programs which will use most of the instructions that the simulator supports. The simulator works if the code runs without error and the values written to the registers are the expected ones. Our single cycle implementation runs all three memory files without any problems. The pipeline implementation however will not work on the memory files given because of control hazards. In order to test it, two new files have been made. The "testpipeline" file which is the same as the "add" file, but with a lot of nop (no operation) instructions to avoid control hazards. The other file "testforwarding" creates a data hazard scenario which the pipeline solves by using forwarding.

## Known Bugs

We spent most of our time debugging on the single cycle implementation, therefore it now performs as expected and has no known bugs. The pipeline implementation however has two bugs that we know of. Load and store word instructions are not getting forwarded so a nop after is necessary for it to work. The other bug is that the break instruction stops the program in the Instruction Fetch stage, thus all other instructions currently in the pipeline are not completed before the break happens. To prevent this we need three nop instructions before a break.