INF-2200

Assignment 2

Andreas Berntsen Løvland & Magnus Dahl-Hansen

October 2020

1 Introduction

In this report we describe how we implemented a simulator of the MIPS-processor architecture using the python-language. This was a mandatory assignment with different levels of difficulty, where the best result would support pipelining and could handle data hazards. Our goal for the assignment was to make a functioning single cycle processor and possibly create a working pipeline.

2 Technical Background

The most relevant topics covered in this assignment are single cycle datapath, pipelining, and data hazards.

A single cycle datapath means that all instructions are completed in one clock cycle and that only one instruction is completed each time.

Pipelining is an implementation technique where multiple instructions are overlapped and executed simultaneously. A Datapath is divided into different stages and instead of executing only one instruction another instruction may begin when the first instruction has completed its first stage. A pipelined datapath is quite a lot faster than a single cycle datapath since it may allow up to five different instructions be processed at the same time. This means that a pipelined datapath can potentially be up to five times faster than a single cycle datapath.

Data hazards are different kind of problems that might occur in a pipeline. More details about Data hazards are discussed in the “Hazard Handling” section of this report but data

hazards occur in general when the pipeline must be stalled because an instruction is depending on another instruction to finish.

3 Design/Implementation

We solved this assignment by implementing different hardware component, which connected would simulate the mips datapath. Figure 1 shows how the final implementation is set up by the different components, to be able to handle the instruction set in figure 2. To be able to execute the different type of instructions, we added all the lines in the mem-file we are running, which is not comments, and added them into a list. The addresses in the memory file gets added into a dictionary as key-values, while each address' corresponding bit-string gets added into the dictionary in our memory as the value.

The instruction memory uses the address input from the program counter to look up in the memory for the bit-string value stored in the memory. The component then splits the bit string into smaller halves, which get sent out as output from the instruction memory.

The control unit is one of the components in the datapath which receives input from the register file. The part of the bit-string the control unit receives is the opcode. The opcode tells the control unit which type of instruction we are going to execute and based on that input it will send out control signals to other components in the mips datapath. To test this component, we give a bit-string as input to the instruction memory and checked whether the output-values were equal to the values we calculated by hand.

Another output from the instruction memory goes to the register file. The register file receives two inputs from the instruction memory, which tells the register file what register to read data from. The data stored in the registers given as input will be the output for this component. The third input the register file receives from the instruction memory tells the register file which register it is going to write data to. This input will vary depending on if the current instruction to be executed is R-format or not. When we implemented testing for this component, we made the register file write values to some of its registers, and the outputting the values from these registers after. If the values, we first wrote to the register file came as output the test would be passed.

The sign-extend component is used for converting a bit-string of 16 bits to a bit-string of 32 bits. The sign-extend components check what value the most significant bit of this 16-bit bit-string is and adds 16 bits of the same value as the most significant one. To test this component, we gave two input values, one with 1 as the most significant bit, and one with 0 as it, and checked if input and output were the same value.

The alu-control unit is a component that sends a control signal to the alu. The control signal it sends out decides what type of operation the alu is going to perform on its input. If the instruction currently to be executed is an R-format instruction, the alu-controlunit also bases its output control signal on the input it gets from the instruction memory, and not just the control signal it receives from the control unit. To test the alu control unit we gave it a control signal like both R-format instructions and I-format instruction. We checked if the control signal output from the alu control unit were correct, according to the control signal input, and we also tested the control signal output for R-format instruction, where we gave it different input values.

The alu component receiver two inputs, one from a multiplexor, and one from the register file. The alu control unit sends a control signal to the alu which tells the alu what operation to perform with the two input values. Output from the alu is the result when the operation has been done. The alu also output a control signal, which is set to high if the two inputs are equal and set to low otherwise. To test the alu we gave two input values to the alu, and a control signal to indicate what operation to perform. We tested the and-, or-, add-, sub-, and nor operations, and compared the output with our own calculations. We also tested the output control signal, testing both with same input and different input.

The data memory component receives the input from the alu and the second data output from the register file. The data memory also receives two control signals from the control unit. The control signals tell the data memory whether to read or write from/to memory. If the read signal is high, the input value on the input address will be output (if address has no value output is 0), and if write signal is high we write the data from the register file to that memory address. To test if the data memory worked as intended, we gave it different address- and data values as input, which it wrote to the memory, and later read the values of the same address. By doing this we wrote a value to the memory, and later read data from memory, and checked if those values matched.

We have two “shift left by two” components implemented. The input value it receives is being shifted two times to the left, which is equivalent to multiplying the input value by 4.

To test this component, we gave a small value as input, and checked whether the output was input*4.

As we must handle the load-upper-immediate instruction, we implemented a shift left by 16 components. This component takes an input and bit shift the value by 16 bits to the left. The value this component calculates is only used if the instruction to be executed is specifically a lui-instruction, as that is the only time when the multiplexor this output is sent to is going to let that value through. To test the shift left by 16 component, we checked whether the input value we gave to component, would be input * 2^{16} to check if it worked correctly.

We implemented an and-gate, an or-gate and an inverter to be able to execute branch on not equal, and branch on equal instructions. The branch equal control signal from the control unit goes as input to the and-gate, together with the zero-signal from the alu. The zero signal from the alu also becomes input in the second and-gate, except that the zero signal in this case is inverted. The other input for this and-gate is the bne control signal. The output from both these and-gates goes to the or-gate, and the result of the or-operation becomes the control signal for a multiplexor. For testing of the and-gate, the or-gate, and the inverter we gave 0’s and 1’s as input values, and checked if the output would be correct according to the truth table for those operations

In the simulator, for every address executed we check if the bit-value on that address if equal to 13. If the value is equal to 13, we print the registers and the memory, and exits the code. This is because the bit-value of 13 is the break instruction, so the program will have to stop itself. The amount of cycles that has been run is also getting printed when the program exits, either by executing a break instruction, or by overflow.

The tick function is the mipssimulator has been modified to check for overflow each time a component is giving an output. All the components in the datapath are added into a list which is getting iterated over, calling for the four functions which reads input, writes output, reads control signals, and output control signals.

For implementing the pipeline, we have created four classes, each class represents one pipeline-component in the mips datapath. Each of the components output the same values

they get as input, also for control signals. They are implemented in the datapath as shown in figure 3. The components are divided into 5 different stages between the pipelines. Each of the five stages has their own lists where the components are added, and we iterate over the lists and call for the four main methods for reading input/control signals and writing output/sending out control signals. Each stage iterate over the objects in its corresponding list, calling the methods for that element, and calls for the method performing the operations in the previous stage (for example, when all components in stage 4 has been iterated over, we start iterating over all the components in stage 3, and so on).

Figure 1: Overview of the single cycle datapath

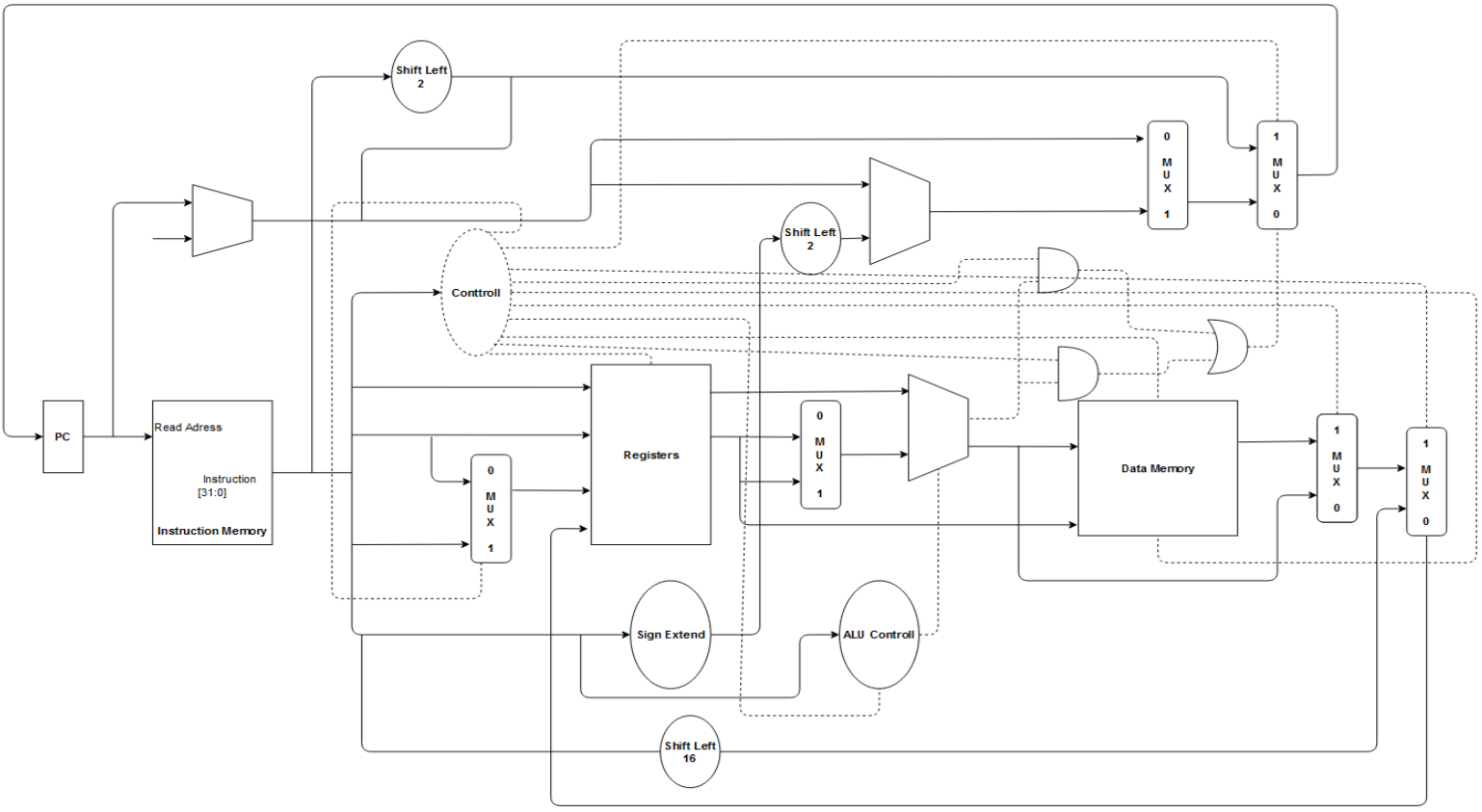
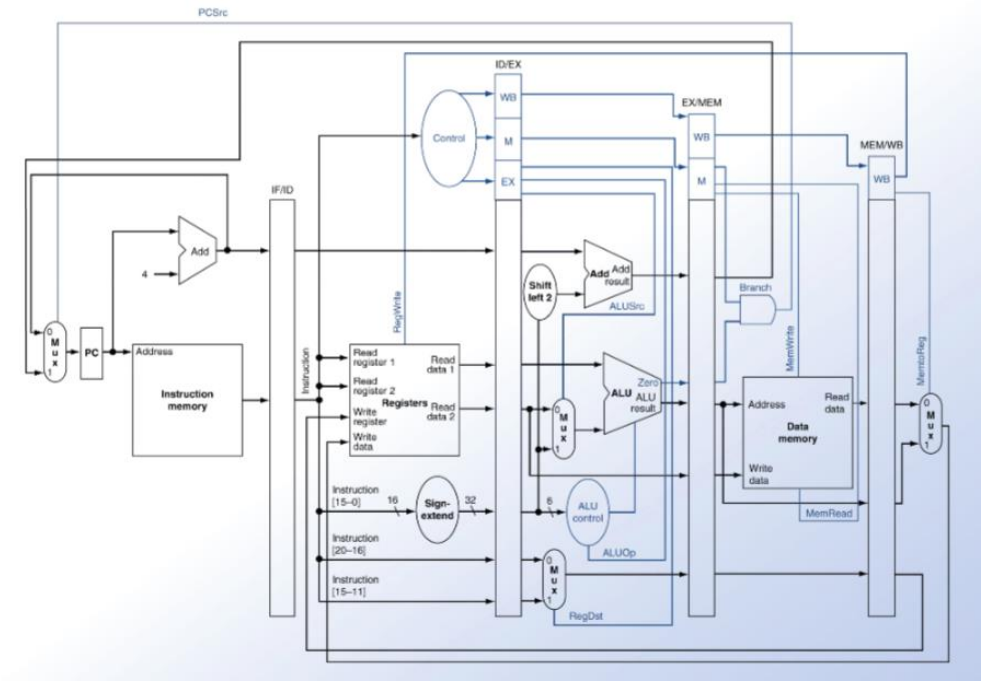


Figure 2: Implemented MIPS instruction set

j – Jump
beq – Branch equal
bne – Branch not equal
lui – Load upper immediate
slt – Set less than
lw – Load word
sw – Store word
add – Add
addu – Add unsigned
addi – Add immediate
addiu – Add immediate unsigned
sub – Subtract
subu – Subtract unsigned
and – Binary AND
or – Binary OR
nor – Binary NOR
break – Break execution

Figure 3: Pipeline overview (not included bne- and lui instruction solution)



4 Hazard Handling

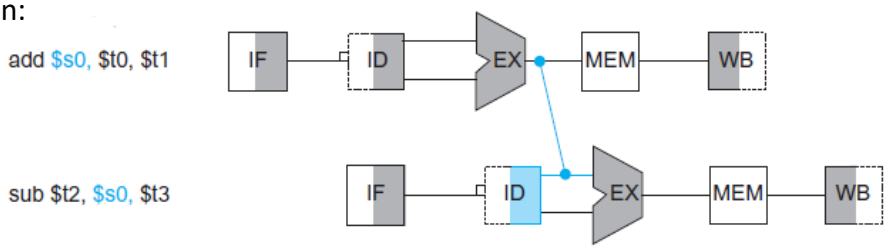
Data hazards occurs when the pipeline must be stalled because an instruction is depending on another instruction to finish first. There are three different types of data hazards named: “read after write”, “write after read” and “write after write”.

The first data hazard, “write after read”, occurs when one instruction tries to read a value from a register which another instruction is trying to update. Say we have an add instruction which is going to write its result to register “t0”, and in the same pipeline another add instruction is trying to read a new value from the same register. So, if the second instruction tries to read a new value from “t0” before the first instruction has a chance to write to it, we have a “read after write” hazard.

The second data hazard, “read after write”, occurs when one instruction tries to write to one register before the previous instruction is done reading from it. This can happen when one instruction is writing to a register very early in its pipeline before the previous instruction has had a chance to read from it. This hazard requires some odd instructions but if you have some instructions that takes a long time then “read after write” hazards can occur.

The last data hazard, “write after write”, occurs when one instruction tries to write to a register after the previous instruction wrote to the same register. An example of this hazard can be that the second instruction completes its task and writes its result to a register while the first instruction is still doing its computation. If this occurred, then the first instruction will finish, and it would overwrite the results of the second instruction.

There are different ways of handling data hazards. One way is to insert no-operations also called “nops” into the pipeline. By inserting nops to the pipeline we simply tell the second instruction to wait so that the first instruction can complete its task and provide the correct result. But even though this way of handling a data hazard works, it is very time consuming since it means adding holes to the pipeline. Another more effective way is to use “Forwarding”. The concept behind this type of hazard handling is to retrieve the missing result from inside the pipeline rather than waiting for it to arrive from registers or memory. Say we have a subtract instruction that needs a value which is currently being processed by an add instruction. One way is to wait until the add instruction writes its result to the correct register OR we could fetch the correct value after the execute stage inside the pipeline. Here is a visual representation:



Forwarding becomes more complex do perform if there are multiple results to forward per instruction or if it is necessary to write a result early on in instruction execution.

A different type of hazard is a control hazard or also known as a branch hazard. This hazard occurs because the processor does not know the outcome of a branch when it needs to insert a new instruction into the pipeline. A branch instruction is executed in the memory stage of the pipeline which means that it takes effect at the end of the pipeline. This means that the program counter will continue to fill instructions into the pipeline which may not be executed and will affect the flow of the pipeline.

Inserting no-ops is an effective but time-consuming fix to this hazard but using “branch prediction” is a better way. Branch prediction works by making a well-educated guess about which instruction to insert to the pipeline next and in this case inserting a no-op will only be needed if an incorrect guess is made. There are two types of branch prediction, called static and dynamic prediction. Static prediction is to always expect one of the ways the branch will take and feed the pipeline with the corresponding instruction, which may lead up to a 50 percent chance of success given there is only two ways the branch will take. There are many subcategories of dynamic prediction, but to make it simple dynamic prediction works by looking at what the instruction has been doing while it is running. For example, it may look at what happened to an instruction the last time it was executed and make a decision based on its outcome.

5 Discussion

Any advantages or disadvantages with your design? Anything that was surprising to you?

The report should contain the following:

1. Summary of known bugs and problems

One known bug occurs when running the selection sort algorithm through our processor. The given numbers are sorted correctly except for two negative numbers. This is caused because the “set on less than” instruction executed in the ALU only compares binary numbers and does not consider if a number is positive or negative. In this case it does not consider that if the leftmost bit is a “1” it means that the number is negative, instead it interprets it as a larger number.

The pipeline implementation is not finished. The pipeline objects are supposed to store data in its registers between the different stages, but these registers have not been implemented yet. Another part of the pipeline implementation which has not been done yet, is the connecting between the other components and the pipelines. The pipelines are connected to the other components, but the other components are not connected to the pipelines.
We have not started on the hazard handling for the pipeline, which means that the main loop is not implemented for handling it, and we have not either implemented the forwarding unit, nor the hazard detection unit. However, we have touched on how hazards occur, and how to handle them.

6 Conclusion

In this assignment we have created an implementation which simulates the MIPS-microprocessor architecture. The current working implementation is a single cycle, but the pipelined version is not far from complete. We had not enough time to start on the hazard handling.