INF-2200-exam1

Sigurd Uhre

October 2021

1 Info

Name: Sigurd Uhre
Email: sigurduhre@gmail.com. suh000@uit.no.
GitHub username: siguhr
GitHub repository: https://github.com/uit-inf-2200-f21/home-exam-1-siguhr
Software: This exam is solved using: Linux Ubuntu 20.18, Visual studio code, Python 3.8.16 64-bit.
Reference style: Vancouver.

2 Introduction

In this exam we will implement binary code simulator for a subset of the MIPS architecture. We will simulate the pipelined datapath and controller mentioned below. Our simulator should support the following instruction and controller:

Instructions: (j:jump), (beq:branch equal), (bne: branch not equal), (lui: load upper intermediate), (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) and (break: break execution)

Controller: (RegDst: Register destination number), (Jump: Jump signal), (Branch: Branch), (MemRead: Memory read), (MemtoReg: Memory to register), (ALUOp; Alu option), (MemWrite: Memory write), (ALUSrc: Alu source) and (RegWrite: Register write).

3 Technical background

Clocking Methodology
The clock-cycle is based on the CPU clock. This clock is feeding constant waves of pulses that are separated into high signal and low signal sent to the microprocessor. The combination of the high and low signal interrupts the CPU at regular intervals and thereby deciding the time between the OS and the CPU-elements.[3]. This cycle is the basis for the pipeline implementation and the way we fetch instructions and and pass them trough our mips-processor.

Pipeline
Pipelining is an implementation technique where the concept is that multiple elements overlap each other when they are executed. Unlike single-cycle where each element needs to start and finish before the connected element(s) get started. The concept is that no element (needed for instruction) stays inactive trough each clock-cycle, which means that the element is either feed the input data or feeds out the output data to the next element. [1]:[*Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 202. s.285].* Pipelining is divided into stages where each stage is connected to the next stage trough a register. This register purpose is to hold the combinational data in the first step-cycle, and put the same data as output to their target destination in the next step-cycle. [2]

Pipeline Hazards
Wen pipelining is implemented, there may be events where the next instruction is not able to execute in the next clock-cycle. These events are called hazards and we divide them into three separate categories: structural-, data- and control-hazards.[1]:[*Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 202. s.290].*

4 Implementation

Datapath
The starting address is feed from PC to instruction memory and the first adder, where the address is incremented by four and further feed to if/id element. Instruction memory collects the address instruction from memory and feeds it to if/id. This elements sends out a 32-bit-string or fragmentation of it depending on the destination. The control unit is feed the full bit string and will determine which control signal to activate and which to keep at the original inactive low signal, depending on the bit-string opcode. The register will be feed two separate fragmentation of the opcode, read register one is given instruction [25-21] and read register two is given instruction [20-16]. If control signal [RegWrite] is activated, the signal notifies the register file that there will be resulting data that needs to be written in to a register. The first input to the register file (instruction [25-21]) indicate which registers data will be read and used as the first argument passed to the ALU trough id/ex.[5]

R-type instruction
For R-type instructions, instruction [20-16] indicate which register data will be read and used as the second argument passed to the alu trough id/ex and mux, the [ALUSrc] control signal is set to low for this type of instruction. Since R-type instructions activate the control unit to send a high signal for [RegDes], the register mux will indicate which register will have the results of the operation write to register, when calculated. The

16-bit immediate-field is then sign-extended to make it 32 bits, feed through id/ex and then feed to the alu-control element. Here the six last significant bits signals that an additional operation is needed to the alu. This is the function field of the instruction and the difference in it gives the alu different instructions for calculations of the two inputs. This field is the only thing that differs the R-type instructions. [5]

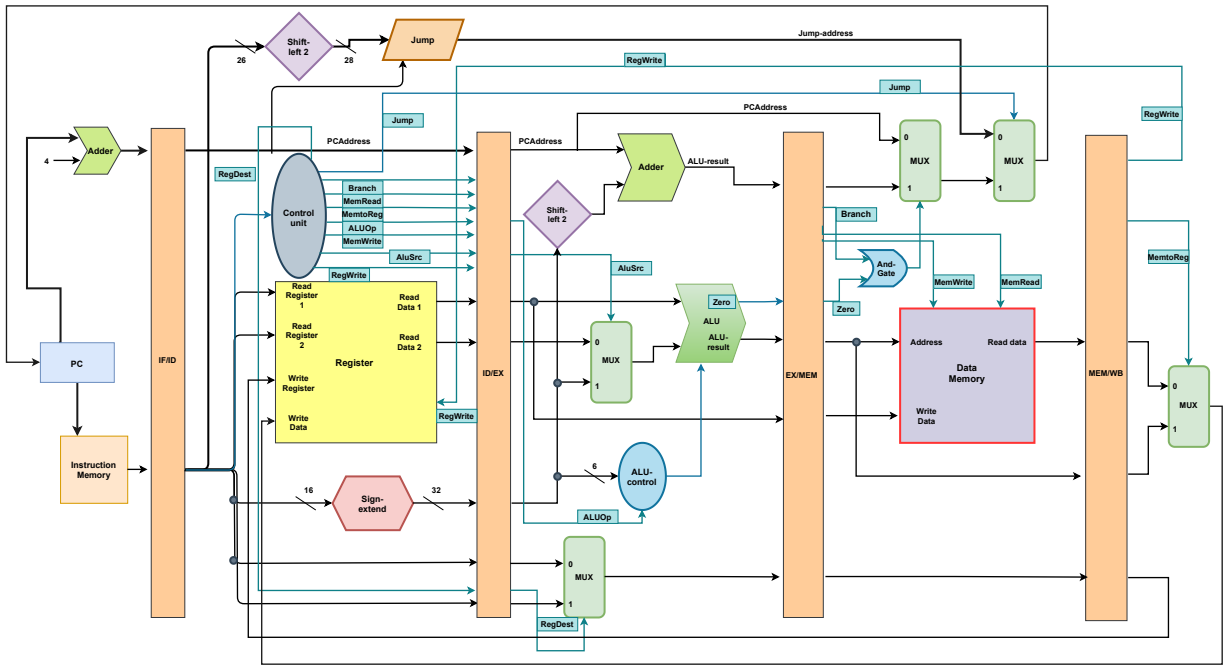

Figure 1: Pipelined datapath for MIPS processor

The alu-result is passed to the result-mux through ex/mem and mem/wb, we now have the data that needs to be written and the register that the data need to be written to. Since [RegWrite] was activated, the data will be written to that register.[5] .The fragmented instruction [20-16] and [15-11] is like mentioned above, passed along to the id/ex from register file, while instruction [15-0] is passed trough the sign extend element where it is altered before arriving at id/ex. if/id passes the new incremented address directly to the id/ex component. The control signals high/low frequency is also passed on to the id/ex, excluding signals that have reached their destination component like [Jump] for jump instructions].

I-types instructions
For I-types instructions excluding [beq] and [bne], the [ALUSrc] control signal is set to high frequency. The altered 16 bit string from sign-extended will be passed trough id/ex and presented as the second argument to the ALU trough the mux, this is the case for load and store instructions. The first input to the register file (instruction [25-21]) indicate which registers data will be read and used as the first argument passed to the ALU trough id/ex. For branch equal [beq] and branch not equal [bne] the [Branch] control signal will be activated and the 16-bit immediate-field from sign extended is passed from id/ex to the shift-left-two elements which feeds the second address adder its second argument, this component (adder) alter the new address and feeds it to the ex/mem component. The ALU will be feed both inputs from the register file trough id/ex like R-type instructions for [beq] and [bne].

For the load and store I-types instructions, Instruction [20-16] will be read and used by the register file, passed on through id/ex and ex/men and presented as the second argument in instruction memory. The alu will produce a result based on the two inputs and the signal from the alu-control. The result is passed on through ex/mem and presented to the address input of instruction memory. It represent a address that specifies to the memory unit the location to store or load the desired data. Depending on the instruction, [MemWrite], [MemRead] and [MemtoReg] will be activated. All the instructions except [jump] activate the [ALUOp] signal when their opcode is read in the control unit. This means that they all need to make calculations in the ALU, but one signal is not enough to differ between which calculations the instructions need the ALU to execute. The control unit have several different [ALUOp] signals (e.g. ”00”, ”01 ”10” with more) that it sends out depending on the instruction. One of these signals is sent to the alu-control element, which interpret the signal and sends out a specific signal value to the ALU. For R-type instruction there is a second element involved in the interpretation of the [ALUOp] signal. For these instructions the [ALUOp] signal ”10” is sent from control unit, the six last

significant bits (function field) of the altered sign-extended instruction [15-0] trough id/ex is also sent, read and interpreted. The alu-control unit sends out specific signal value to the ALU based on the differentiation of the function field combination despite the fact that their control signal is the same. [5]

If the ALU is required to make calculations, the two input values along with the specific signal value from alu-control [ALUOp] determine the ALU result. The ALU preforms different calculations for the different instructions, this include not only logical calculations but also comparison. For [beq] e.g. the two data inputs needs to be compared. If the condition for this instruction did not pass, the address will just be incremented by four and the rest of the instruction will not be utilized. If the condition passes, the ALU will then send a zero signal to the andgate through ex/mem and further on to the connected mux signifying that the condition passed. The andgate is also half activated by the high signal from the branch control signal. Since the condition passed the PC needs to be loaded with the address that the adder assigned to it. The 16-bit immediate-field is then sign-extended to make it 32 bits, feed through id/ex and then shifted left twice. This number is then added to the incremented address of the current instruction in the second adder, before it is feed to the pc as the next address. [5]

J-type instruction
For a J-type instruction (jump) the opcode is feed to and read by the control unit, setting [Jump] to a high signal and all the other components to a low signal. This signal "activate" the mux that is to be used by the "jump" instruction. Since a "Jump" uses no registers, no memory and preforms no ALU calculations, no other components needs to be used, leaving "Jump" as the only high signal. Like all other instructions, the address is sent from pc to adder where it is incremented by four and feed to the if/id. The remaining bits [25-0] needs to be 32-bits in length and at this point there are only 26 remaining in the instruction. The new incremented address is feed from if/id to the jump element where the four most significant bits are truncated (cutt opp), reused from the previous address and given as the first argument to the jump element. Bits [25-0] is shifted left twice in the shift-left-two component and then feed to the jump element. The input is truncated to fill in the gap behind the four most significant bits, resulting in a complete address for the next instruction. The address is feed to the pc through the mux activated by the [Jump] high signal. [5]

Simulator implementation
The Simulator is tick based, meaning that for each clock cycle there will be some elements that are active and some that are inactive depending on the instruction and the stage of the cycle. Several clock cycles goes trough before an instruction is fully executed. The elements are connected to each other in the mips simulator file. Each element is connected to one or several components trough datapaths or signal-paths. They will have declared available spots for inputs/outputs data and signal connections. In figure one we illustrate the different elements and their connections with each other trough datapath (black arrows) and signal-path (blue arrows). We make the connection with the connect function, we declare what file we are connecting up to, what file both types of input comes from with their specific names that mirrors the input order in the file. If the element have any output, we declare the specific names of the output in a specific order that mirrors the output order in the file. All the elements are declared in a specific order that the simulator runs trough in order, starting with the increment-four constant and ending with register file.

Functions
The elements hold the data in our processor. They are components that are feed, hold and sends out data to and from connected elements. Each element is represented in a separate python-file which contain its specific mechanism including data and signal input, output, test mechanisms, along with specific calculations if any. One type of component can represent several elements in the processor pathway, the difference is their connections with other elements and thereby their input, output and signals will also differ. Their original structure however, remains the same. The control unit sends out high (active) or low (inactive) control signals to several elements, altering some of their mechanics if activated or continue with inactive mechanics.

Each element implemented (figure 1) provides its own function in the mips-simulator. The if/id, id/ex, ex/mem and mem/wb elements functions as registers where the soul purpose is to hold the combinational data from the previous step-cycle, and put the same data as output to their target destination in the next step-cycle. The pc-adder function is to increment the current address by four. The control unit reads the opcode, depending on the code-combination it turn on or leave off the control signals. The [ALUOp] signal have five combination of on-signal to differentiate the alu-instruction. The register file have one input control signal [RegWrite]. If inactive the register file will fetch the value in register from the register addresses given in input location one and two and put the registers value as output one and two. If active the the same as above happens, but the register file is notifies that there is certain data that needs to be stored in a certain register value trough write back method, and the register is given in input three and the value in input four.

Sign-extend is feed the 16-bit immediate-field and will extend the filed to 32 bits. If the instruction is [lui] load upper intermediate, the extended field will be put in the last significant end and represented as a bit string output. For the other instructions, the extend field will be put in the most significant end and also represented as a bit string output. The shift-left-two element makes a shift left twice operation on the integer value of the bits [6:32], excluding the opcode. The branch adder takes the shift-left-twice integer as the second argument and adds it to the current address. ALU-control differentiate between the different versions of the [ALUOp]-signal and feeds the alu a value-signal that correspond its requested calculations. Depending on the value signal, the alu differentiate what type of operation it is to execute between the two inputs presented. Here we have logical operations like e.g. add and subtract along with comparison operations. Depending on the signal value for calculations and input values the alu will either produce a result value or a Zero control signal.

Data memory will receive one of the signals [MemWrite] or [MemRead]. Depending on the signal the element will either read the memory stored in the input address value ([MemWrite]) or save the input data to a memory address in memory [MemRead]. The andGate send out its control signal if the condition that both input signals at high is met. The jump components takes in to inputs, truncates selected sections of both bit inputs, places them in a specific order and represents its as a output.

5 Methodology

Approach
Our approach for handling pipeline has been to implement stall elements (e.g. if/id and id/ex). They wok as a buffer that operates in-between two clock-cycles and simply store the data as a register trough the cycle-change. Instruction fetch is the if/ part where the register is feed the instruction from instruction memory and the incremented new address from the adder. The next stage is instruction decode (id) which is a part of both the components if/**id** and **id**/ex . This clock-cycle is the if/id-components output and the id/ex-component input cycle, which means is is also the register file and sign-extended output cycle. This concept continues trough the ex/mem and mem/wb components at the clock-cycles continues resulting in a write back to the register file. [1]:[*Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 202. s.290-292, 302-307].*

Cost and comparison
The theoretical comparison between a single cycle versus pipelined is constant for the same instructions. The time between instruction will not be the same for each instruction for bot pipelined and non-pipelined, we must therefore relate to the worst-case cycle for both types, although other instructions will give better results for both. We take a theoretical example: Three lw-instructions with operation time for all the functional units at 200 ps, excluding register file read or write which takes 100 ps. The single cycle needs to complete a full instruction (IF-reg-Alu-data access-reg) before it can start on the next lw-instruction, resulting in a 800 ps cost for each instruction and a total of 2400 ps.*[1]:1. Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 2021. s.287*

The pipelined execution will not wait fro the full instruction to complete before starting on the next. When instruction fetch is completed it will start over and fetch the next address, and pass it on to the next stage which just completed its part of the first instruction, like a conveyor belt execution. This will result in a total of 1400 ps for three load word instructions for the pipeline execution. As 2400 ps divided by 1400 ps equals 1,71, we can say that single cycle is 71 percent more costly in this example. If we resemble the number of instruction in a program, we can add 1 million instructions to both methods giving pipeline a total of 1 million times 200 ps plus 1400 ps or 200,001,400 ps. For non-pipeline this would be 1 million times 800 ps plus 2400 ps or 800,002,400 ps. Giving a ratio of 4, which is close to the ratio between the instructions: 800 ps divided by 200 ps = 4. *[1]:1. Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 2021. s.288-289* . Pipelining increase the processors performance by making better use op the components by overlapping the instruction execution.

Hazards
Data hazards occur when a instruction reads a register in the register file that a previous instruction overwrites with the write back input in a cycle that happens later. If we do not eliminate these type of data hazards the pipelining will produce incorrect results. One way to eliminate this type of hazards is by the stall approach. The stall approach stalls the instruction memory if it detect any data hazards, instruction memory do not read any new instructions until the stall is resolved. To detect the hazards condition we need to implement a function that detects the Write Register in the pipeline register files id/ex, ex/mem and mem/wb. If the write register input in register file is equal to one of the two read register inputs, the stall function will occur. [4]

Control hazards occur when the pipeline makes mistakes while branching. When we branch to a new instruction the instructions seated behind this branch instruction in the pipeline becomes invalid. One way to solve this type hazard is to include a function that can calculate the branch instructions offset in the instruction decode cycle. By forwarding the data we can execute the branch one cycle later. [4]

Single-Tests
To check if the specific element works we can implement a test. The idea is to create the tests before finishing the element, so that we know that the element works as it supposed to. We test the setup, meaning that we test the way the element is connected with another element by creating test-input connections and test-output connection, we also test the self-elements. This setup resembles the way the element is connected with others in simulator connections, so if the elements i correctly implemented and test is passed, we know that this implementation work with this setup. The problem is that we have made corrections to the files after the tests have been created to solve errors which occurs when running the entire simulator. The test for some files have not been updated and for some files we did not simply prioritize them. In instruction memory we have a successfully design a test that a scenario input for load word and store word instruction and a MemRead or MemWrite high-signal. We test if the output data equals the instruction in memory for this address for the lw-instruction. We test if there is a memory address where we can store the new data in the memory address.

6 Results

Test
We where not able to implement data or signal hazard handling. Our processor will be vulnerable to both data hazards and signal hazards. If we run the given test "python3 -m pytest -rPf" we get a total of 16 failed and 16 passed sub-test within the pytest. The sub-tests that passes are the one where zero out of zero points are the desired output, with the exception of the first sub-test. All the test that fails are the one where there are points to be made, and we get e.g. zero out of two/one or four points. If we run run the simulator in the source file, we can detect that there is more than a few bugs and problems involving that no data is is stored in the register, most likely resulting in several consequential errors.

Simulator test with the mem-files shows that our break instruction only works for the add.mem file. Selection-sort instruction memory crashes after the beq-instruction because it is not represented a valid address fro the mem-file. The Fibonacci file also crashes after the beq-instruction is completed, and instruction memory is represented a non valid address. This sis an indication that there is something wrong with the way our processor handles beq-instructions, involving the sign-extend-, shift left twice- and branch adder-element. The branch-,break- and lui-instructions are most likely not supported correctly in this implementation.

7 Discussion

The elements seems to be connected properly, with data- and control-signals arriving at the desired destination. The pipeline structure- implementation seems to be implemented correctly. However, since we know that there is no data being stored in the register, consequential errors is inevitable. The source file test gave 16 failures and 16 passed test where none of the sub-test that required points was passed. It is difficult to determine the type of instructions that are properly supported in this process. Based on the errors we get, it will be relevant to suggest that the functions that depend on sign-extend do not work correctly.

8 References

1. Patterson D, Hennessy J. Computer organization and design mips edition. 6th ed. 2021.

2. Concept of Pipelining — Computer Architecture Tutorial — Studytonight [Internet]. Studytonight.com. 2021 [cited 28 October 2021]. Available from: https://www.studytonight.com/computer-architecture/pipelining

3. Definition of clock [Internet]. PCMAG. 2021 [cited 28 October 2021]. Available from: https://www.pcmag.com/encycloped

4. Hazards [Internet]. Massey.ac.nz. 2021 [cited 28 October 2021]. Available from: https://www.massey.ac.nz/ mjjohnso/note

5. [Internet]. Youtube.com. 2021 [cited 29 October 2021]. Available from: https://www.youtube.com/watch?v=oETOwVBzu
$ProgressiveLearningPlatform$