Report

This document provides a high-level overview of this software's architecture and its lifecycle. We have documented our approach and thinking for each stage of the software's lifecycle and what each process yielded.

We also go over more difficulties and challenges that we've encountered in more detail in the Appendix.

Prerequisites

This report assumes the reader already has knowledge and understanding of the terms used when discussing the Tomasulo algorithm/architecture, such as reservation stations/buffers, as well as general computer architecture terms/concepts, but all are not completely required in order to proceed.

Table of Contents

Architecture and Design

This chapter documents an overview of our software's architecture and design.

The Tomasulo Algorithm

Due to the heavy dependencies of the features of this program on each other, and how almost every component needs data to flow to it from another component, and the absence of time to draw system diagrams and strictly define how data would flow throughout the system, we decided to start writing down a text description of the tomasulo algorithm and dumped all the details we could think of and the challenges that we would face during implementation and our (possible?) solutions to them.

Here is our written draft of the algorithm. (Please note that this was built upon and further advanced and any other details not mentioned here have been taken into consideration in the actual implementation)

Clock Cycle: -1: Read instructions and load them into Instruction Cache.

0: Fetch instruction from Instruction Cache into Instruction Queue.

1: (Issue & Read Operands)

Before Issuing:

- Decode Instruction.
- If no Structural Hazard (availability of structure to carry out the operation/there's space in Buffer/Station) && In case of Load there is no Store manipulating the same address in the Buffers (eliminating RAW hazard), && In case of Store there is not either a Load or a Store manipulating the same address in the Buffers. (eliminating WAR and WAW hazards), Otherwise; instruction won't be issued and will remain in the Instructions Queue.

  Then, - Issue Instruction to its Buffer/Station, - Set the Buffer/Station to busy, - If not a Store, Update the Qi field in the Register file for the Instruction's Destination Register with the tag of the Buffer/Station that the Instruction has been added to, - Add to a Hash Map ==> Key: Tag of the Buffer/Station, Value: Cycle Number that it's issued in (tracking when each instruction was issued exactly).

  If Load instruction, skip the next check (as effective addresses are assumed already given so no need to check registers):

Check Operand Registers in the Register File:

- If the Operand Registers are free (their Qi = 0) then copy the value to Vj/Vk. Else, copy the Buffer/Station tag responsible for this Operand from Qi in the Register File to Qj/Qk.

2: (Execute)

- For each Station, if it has Qj and Qk both equal 0, Execute(Decrement Cycles Left).

- For each Load or Store Buffer, if memory is not busy, Execute (Decrement Cycles Left and set memory to busy).
- Each instruction keeps executing according to its required number of cycles to finish. If it's a Store, wait for the value to be stored to be ready before Executing.
- We will use an array of finishedInstructions in which we will add an Object {tag, value} of the finished instruction to, to later in the next stage be used to select one instruction only to publish its value and tag to the Bus(based on FIFO) using the HashMap in the issue stage (storing the clock cycle in which each instruction was issued). Thus, this array will be sorted based on the clock cycle number that the instruction was issued in. So, if instruction is finished, compute the result and add the {tag, value} to array of Finished Instructions except if it's a store instruction we don't add it to the array as it won't publish anything to the bus, and excluding the BNEZ instruction also as it doesn't write anything to the Bus, it directly update the PC register value if it is successful.

3: (Write Result)

- If Finished Instructions Array is empty, Return.
- Remove an instruction {tag, value} from the Finished Instructions Array And Write that instruction's result to the Bus.
- Find the station/buffer that has the same tag as that on the bus and clear it and unset the station/buffer busy status and for load/store unset the memory busy status as well.
- For each Station: If the Station has Qj equals the tag published on the bus Set Qj to 0. Set Vj to the value from the bus. If the Station has Qk equals the tag published on the bus Set Qk to 0. Set Vk to the value from the bus.
- For each StoreBuffer: If the StoreBuffer has Q equals the tag published on the bus Set Q to 0. Set V to the value from the bus.
- For each Register in the RegisterFile: If the Register's Qi equals the tag published on the bus Set Qi to 0. Set content to the value from the bus.

  (Note: Load Buffers don't need updating as they only have effective address)

Assumptions

We have also made a list of assumptions with regards to the algorithm, hardware architecture, instruction syntax, and more.

Here are our assumptions:

- All events are shown describing what occurs at the of the currently displayed clock cycle.

- The Tomasulo algorithm starts from clock cycle 0, having the first instruction fetched at the end of the cycle.
- Assuming the fetch, write result and the issue stage take one cycle for all instructions, where in the issue stage (the first half decoding occurs, and the second half stations, buffers and register file assignments occurs).
- Regarding the instructions syntax, for adding a label, the label must end with a colon (':'). When executing a branch instruction to jump to this label, the branch instruction should have the label without a colon (':'). For example:

  ```
  LOOP: L.D F1, 0
  BNEZ R1, LOOP
  ```

- Addresses for Load and Store Instructions are assumed already precomputed and the instruction is written with the address as an immediate value.

  So this:

  ```
  L.D F2, 100
  ```

  Not this:

  ```
  L.D F2, 32(R2)
  ```

- In an instruction, fields following the operation field are separated from each other by a comma and a space.

  ```
  R2, R3
  ```

- If two instructions finish at the same time and want to write to the bus, we pick the one which came first. (FIFO)
- When issuing a BNEZ instruction in an Add/Sub station, we put the operand register (whose value we evaluate against 0), in Vj if it is not used by another instruction and in Qj if otherwise, and also we put the address of the referenced label in the A field of the station.
- As long as a BNEZ instruction is not done, we cannot fetch no more instructions, aka. a stall occurs (to avoid fetching the wrong instruction).
- No two stores or a load and a store referencing the same memory address can exist in the Buffers. One only must be issued to avoid any kind of memory related hazards.
- Multiple loads and stores can exist in the buffers at the same time as long as they are referencing different memory addresses, but only one load or store is allowed to execute in the memory at a time, no other loads or stores can execute concurrently.
- When it is time for picking which load of the Load buffers or store of the Store buffers to execute, we pick based on FIFO to ensure that no

particular instruction is waiting for a long period, then the memory is marked as busy and other loads and stores must wait for it to finish.

- There are only one type of Adders and Multipliers supporting both integer and floating point instructions, recognizing them based on a control input.
- Only a store/load existing in store/load buffer can execute in the cycle after an executing store (for both load and store) and load (for a store only) writes its result.
- When a structural hazard occurs (all buffers or stations are busy), an instruction can be issued in the cycle after one of the busy stations/buffers writes its result.

The Tomasulo Architecture

To keep track of the real hardware architecture and to properly simulate it, we took as reference a couple of diagrams which can be found under this directory, entitled `architecture_1.png` and `architecture_2.png`. Both are taken from the German University in Cairo's Microprocessors Course for the year 2023, by Dr. Milad Ghantous.

Application Data Flow

Our software uses a client-server architecture.

The client takes the inputs from the user like instructions, instruction latencies, buffer sizes, etc. - which are necessary to configure a run of the Tomasulo algorithm - then makes a HTTP request to the server using these inputs and waits for the response. (See `Appendix` for an interesting story as to why we chose client-server)

The server then proceeds to run the algorithm using the inputs from the received HTTP request, records the required values at each clock cycle such as station and buffer values, and returns them in an array to the client in an HTTP response.

The client then takes over with the simulation, allowing the user to cycle through each clock cycle and view a snapshot of the system at the end of that cycle.

OO Design

We have decided to use Object Orientation (OO) as the approach for our back-end logic.

We use classes to simulate the hardware components, and interfaces and abstract classes to better organize code dependencies. Here are some of our classes:

- `ReservationStation` (Abstract Class)
- `AddSubReservationStation` (Concrete Class)

- `Buffer` (Abstract Class)
- `LoadBuffer` (Concrete Class)
- `Executable` (Interface)
- `DataCache`
- `InstructionCache`
- `InstructionQueue`
- `CommonDataBus`
- `RegisterFile`

We also have Handler classes to carry out the logic of each stage in the Tomasulo pipeline. Some of those classes are:

- `FetchHandler`
- `IssueHandler`
- `ExecuteHandler`
- `WriteHandler`

Finally, our `Tomasulo` class, which is the main class, contains our program's main loop that runs until the input program instructions end, and uses the other classes to run the full pipeline stages every cycle.

Technology Stack

Our language of choice was either going to be Java or TypeScript due to our familiarity with them and the availability of OO features in both of them. Since we knew we would develop a GUI, we opted for TypeScript so we can use one language for both frontend and backend.

We used React with Typescript and Material UI (MUI) styled components to create our GUI. Other than that, we did not use any other fancy technologies.

Implementation

This chapter documents the implementation of our software.

Overview

The first thing we implemented was the inputs page for users to input the program configurations, and then we directed all of our effort to the backend. After we were done implementing the backend and testing it manually in the CLI (more on testing in the `Testing` chapter), we proceeded to implement our GUI to display the output and refined the input page.

Backend

We began visualizing how our backend would work by referencing our written algorithm and our hardware overview diagrams to maintain as realistic of an implementation as possible. (This also led us to an interesting problem on simulation realism, see more in the `Appendix`)

We wrote the majority of our backend code in group coding sessions as the tasks could not be split across the team members due to the heavy dependencies.

We spent much of our time refactoring and cleaning up after implementing each feature as well as thinking about how data would flow to and from it, how it relates to other components, where it sits in the system overview and any edge cases that would cause it to break. This dedication minimized the presence of bugs in our codebase ***significantly*** and made testing quite simple, and whenever we did find a bug, we could easily fix it or integrate a feature that was missing because our code's design allowed for it.

Frontend

We implemented our frontend using React.

We have two pages, or rather two "big" components that you can consider to represent pages, one for getting the inputs and the other for displaying the outputs.

We did not need to use any routing, as we only needed the root route `/` in which we initially rendered the inputs page, and then rendered the outputs page in the same route after the inputs are submitted.

Our client-side data was passed around using React Context, and validated using react-hook-form and zod. Instructions are MIPS x64 instructions and are either inputted in a text area or in a uploaded in a .txt file, and in either way they are parsed to an array of instructions.

Overall, our frontend was fairly straightforward to implement, and most of the complexity was in the backend.

Code Structure

Our codebase is split into two main folders (excluding the docs folder and any other non logic-related folders), those being `client` and `server`. These two would be deployed separately but, in our case, we just run in the localhost, so we can comfortably keep them both under the same repository and run each of them on their port on the localhost and have them communicate via HTTP normally.

Here is our project's file tree with some omissions for simplification.

```
.
  client
    sample_programs
    src
    components
      input
| |       <---------------- Input components
      output
```

```
| |    <----------------- Output components
   constants
   contexts
   interfaces
   schemas
   types
   utils
  docs
  server
  src
  constants
  interfaces
  main <----------------- Main backend logic, containing all the classes
   arithmetic_units
    AluElement.ts
    FPAdder.ts
    FPMultiplier.ts
   buffers
    Buffer.ts
    LoadBuffer.ts
    StoreBuffer.ts
   caches
    DataCache.ts
    InstructionCache.ts
   reservation_stations
    AddSubReservationStation.ts
    MulDivReservationStation.ts
    ReservationStation.ts
   tomasulo_stages <----------------- Main pipeline stages along with other minor helping s
   clear
    ClearHandler.ts
   execute
    ExecuteHandler.ts
   fetch
    FetchHandler.ts
   issue
    DecodeHandler.ts
    IssueHandler.ts
   update
    UpdateHandler.ts
   write
    WriteHandler.ts
   CommonDataBus.ts
   InstructionQueue.ts
   RegisterFile.ts
   Tomasulo.ts <----------------- Main class holding the main program loop
  types
    enums
  utils
  index.ts <----------------- Server file
```

Workflow

We maintained a constant feedback loop of incrementally adding features and refactoring after adding each one of them to keep the codebase clean and maintainable.

We used GitHub Issues to track bugs, necessary features and questions.

Testing

This chapter documents the testing phase of our software.

Running the Code

Early on, to run our backend, we created a server that just runs the algorithm once we run the server and then exits. We added console logs throughout our codebase to test the output in each component at every cycle and evaluated the outputs.

This was enough for us initially when testing using the CLI. However, we later transitioned to an express server to be able to listen on a port and have the client make requests to the server to run the algorithm and retrieve the results to display in the GUI.

Test Cases

We tested the correctness of our codebase by running many test cases. The following are some of them.

Test Case #1: It was a comprehensive test case covering every corner case, as it included loops, data hazards, structural hazards, hazards regarding having multiple loads and stores referencing the same memory address.

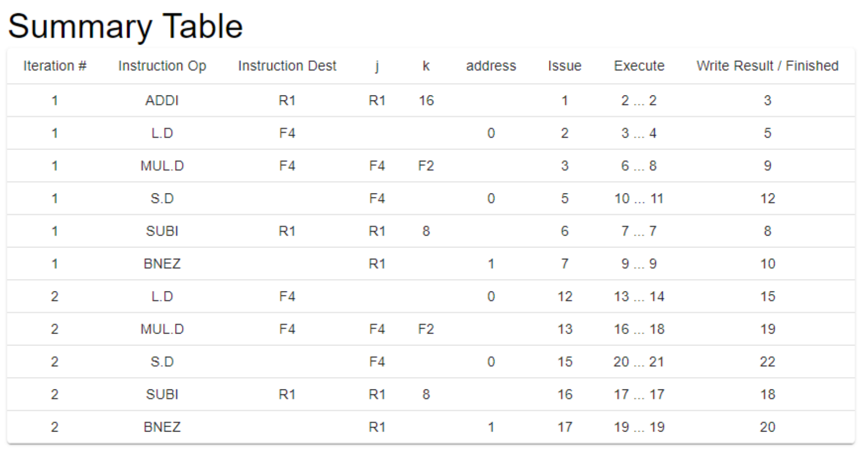
```
ADDI R1, R1, 16
LOOP: L.D F4, 0
MUL.D F4, F4, F2
S.D F4, 0
SUBI R1, R1, 8
BNEZ R1, LOOP
```

Instruction Type	Latency
`FP Add`	2

Instruction Type	Latency
FP Subtract	2
FP Multiply	3
FP Divide	3
Int Subtract	1
Load	2
Store	2

Buffer	Size
Load	2
Store	2

Reservation Station	Size
Add/Sub	2
Mul/Div	2



Summary Table

Iteration #	Instruction Op	Instruction Dest	j	k	address	Issue	Execute	Write Result / Finished
1	ADDI	R1	R1	16		1	2 ... 2	3
1	L.D	F4			0	2	3 ... 4	5
1	MUL.D	F4	F4	F2		3	6 ... 8	9
1	S.D		F4		0	5	10 ... 11	12
1	SUBI	R1	R1	8		6	7 ... 7	8
1	BNEZ		R1		1	7	9 ... 9	10
2	L.D	F4			0	12	13 ... 14	15
2	MUL.D	F4	F4	F2		13	16 ... 18	19
2	S.D		F4		0	15	20 ... 21	22
2	SUBI	R1	R1	8		16	17 ... 17	18
2	BNEZ		R1		1	17	19 ... 19	20

Figure 1: Test Case 1

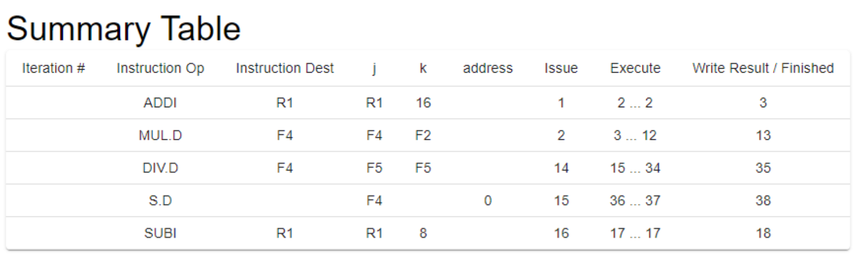
Test Case #2:

```
ADDI R1, R1, 16
MUL.D F4, F4, F2
DIV.D F4, F5, F5
S.D F4, 0
SUBI R6, R5, 8
```

Instruction Type	Latency
FP Add	3
FP Subtract	3
FP Multiply	10
FP Divide	20
Int Subtract	1
Load	2
Store	2

Buffer	Size
Load	1
Store	1

Reservation Station	Size
Add/Sub	1
Mul/Div	1



Summary Table

Iteration #	Instruction Op	Instruction Dest	j	k	address	Issue	Execute	Write Result / Finished
	ADDI	R1	R1	16		1	2 ... 2	3
	MUL.D	F4	F4	F2		2	3 ... 12	13
	DIV.D	F4	F5	F5		14	15 ... 34	35
	S.D		F4		0	15	36 ... 37	38
	SUBI	R1	R1	8		16	17 ... 17	18

Figure 2: Test Case 2

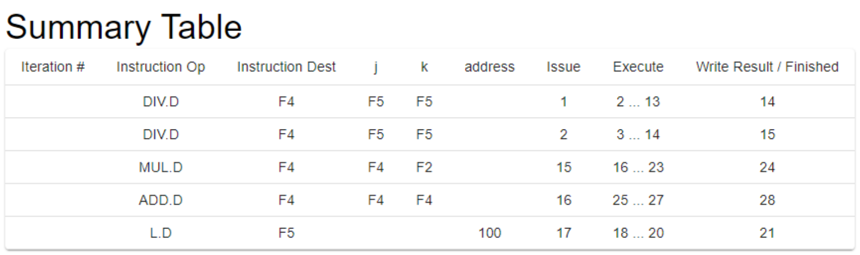
Test Case #3:

```
DIV.D F4, F5, F5
DIV.D F4, F5, F5
MUL.D F4, F4, F2
ADD.D F4, F4, F4
L.D F5, 100
```

Instruction Type	Latency
`FP Add`	3
`FP Subtract`	5
`FP Multiply`	8
`FP Divide`	12
`Int Subtract`	1
`Load`	3
`Store`	3

Buffer	Size
`Load`	2
`Store`	2

Reservation Station	Size
`Add/Sub`	2
`Mul/Div`	2



Summary Table

Iteration #	Instruction Op	Instruction Dest	j	k	address	Issue	Execute	Write Result / Finished
	DIV.D	F4	F5	F5		1	2 ... 13	14
	DIV.D	F4	F5	F5		2	3 ... 14	15
	MUL.D	F4	F4	F2		15	16 ... 23	24
	ADD.D	F4	F4	F4		16	25 ... 27	28
	L.D	F5			100	17	18 ... 20	21

Figure 3: Test Case 3

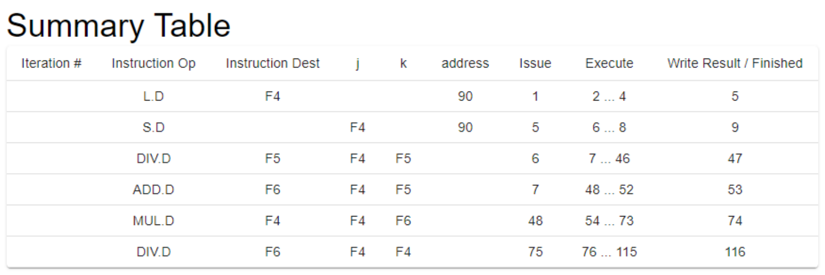
Test Case #4:

```
L.D F4, 90
S.D F4, 90
DIV.D F5, F4, F5
ADD.D F6, F4, F5
MUL.D F4, F4, F6
DIV.D F6, F4, F4
```

Instruction Type	Latency
FP Add	5
FP Subtract	7
FP Multiply	20
FP Divide	40
Int Subtract	1
Load	3
Store	3

Buffer	Size
Load	1
Store	1

Reservation Station	Size
Add/Sub	1
Mul/Div	1



Summary Table

Iteration #	Instruction Op	Instruction Dest	j	k	address	Issue	Execute	Write Result / Finished
	L.D	F4			90	1	2 ... 4	5
	S.D		F4		90	5	6 ... 8	9
	DIV.D	F5	F4	F5		6	7 ... 46	47
	ADD.D	F6	F4	F5		7	48 ... 52	53
	MUL.D	F4	F4	F6		48	54 ... 73	74
	DIV.D	F6	F4	F4		75	76 ... 115	116

Figure 4: Test Case 4

Appendix

This chapter provides extra details, obstacles and challenges that we ran into, should you be interested.

The N-1 Bug

Initially, we were not going to separate our application into a client and server, and we were just going to have frontend and backend logic all in one place with no need to make network requests between the frontend and backend, which was - and still seems like - the obvious, intuitive way to go about an application like this.

However, we ran into an absurdly rare bug in which we tried setting a value of a field in an object to a number, and what happened was that number would always get set as that number minus one. No matter what number N we gave it, it would give us back N-1. We tried everything, and the error made absolutely no sense. We console logged the value inside the setter for this object's field, and it was N inside the setter, even after setting the value, but then after exiting the setter method and going back, and console logging in the next line directly after calling the setter on this object, the value would be N-1.

The only way that it was fixed was when we separated our backend logic into a server and called that server from our React client.

We reached a conclusion that this bug is due to some 0.00001% chance error (probably an exaggeration, but you get the point) that is caused by the Typescript compiler due to some unusual arrangement of dependencies in our project.

Due to our limited time, we decided not to delve deeper into investigating how to resolve this issue and not go searching for answers for weird compiler problems that we were unlikely to come across a ready solution for. Thus, we decided to swallow the pill and stick to client-server architecture.

To Simulate or Not to Simulate

When coding up a project such as this one, there comes a point where the line becomes blurry as to how realistic your simulation should really be.

At the end of the day, we are attempting to simulate hardware components, using nothing but software.

Here's the problem.

> How far do you go in simulating the real thing?
>
> How far is too far?
>
> Where is the point in which you can stop and say this simulation is real enough?

In our case, we reached a point where an Adder, for example, had to write its result to the bus - which is nothing but a bunch of wires to transmit the data. We found ourselves asking whether we should simulate the bus, making a class for the bus and having the components then read from it.

Now, we ended up adding the `CommonDataBus` class, effectively simulating these wires, but one does not have to stop there. We can just keep going on and on and on with no end to this. We could keep simulating down to the level of the logic gates.

Another thing is, many times during this project we found ourselves thinking that implementing the real thing would've been easier, because we have to make up fake scenarios and ways that go against the actual logic of the thing we're actually trying to simulate, because it is just that - a simulation. It's not the real thing, so you often have to go out of your way to do things that go against the real logic in order to make the simulation work.

Perhaps this is all too philosophical. Perhaps it is nonsensical. Perhaps this is just how it is with these kinds of projects.