Computer Archticture
Project 2: Tomasolo's Algorithm

Seif Yehia Sallam
ID: 900193668
e-mail: Seif_Sallam@aucegypt.edu

The American University in Cairo

Under supervision of
Professor: **Cherif Salama**

1 Project Overview

The goal of this project is to implement a Tomasolo's Algorithm without speculation simulation on a 16-bit RISC ISA.

The language used in this project is **C++**.

2 Design Principle

The project is created with ease of control in mind such that it can be used for educational purposes and holds as many features that could be implemented as time could give. This project designed the simulator to support a friendly graphical user interface that is easy enough to work with in any environment.

The coding design principle was imperative with some use of Object Oriented designs. I did not choose to go full Object Oriented as it felt it would over-complicate the process. The different instructions and stations have almost identical approachs when it comes to issuing, execution, and writing back. Since the whole system was designed to run in parallel, a sequential approach with Object Oriented in mind would be difficult to comprehend, yet not so hard in imperativie and procedural manner.

Main Classes

There are five main classes and structs used to create the whole project.

1. Application:
   The main class that handles the events and organizes the interface. It also handles the differnet input/output of the window and is the main comunicator with the backend of the algorithm.

2. Controller:
   The Controller class that controlls the different reservataion stations.

It upades every time we advance to the next cycle where it issues, executes and writes back if available.

3. Instruction:
   A small struct that defines what this ISA instruction consists of. It has its attributes of rs1, rs2, rd, and imm, along side with other important methods such as Parsing the instruction once taken.

4. ReservataionStation:
   A class that serves as a placeholder for the different values a reservataion station can take. It merely executes the instruction it has according to the instruction type.

5. RegisterFile:
   The register file in this implementation is not just an array of 16-bit registers of size 8. It also has the different producing units of the current register and the name of that register. It is created essentially for the Controller to be able to handle differnet situations and hazards.

NOTE: The *InstructionsUnitCycles* class is just a singleton class that defines some global variables that can be used in different places. It doesn't have any methods or purpose other than serving as a namespace. I created it as a class and not a namespace just to keep the design consistant.

3 How the simulation works

Instruction Loading

The first thing that works once the program is opened is Loading the correct instructions from a text file (.txt), the instruction memory gets filled with the different instructions that are parsed. Parsing an instruction is equivelant to fetching and decoding it and knowing the different elements in it. The instruction memory is then used as a main asset to know the current instruction used and worked on it.

Instruction issuing

The user can then start advancing the cycles one by one by pressing the button *Next Cycle*, which will increase the cycle number (starts with -1), and advances the PC. The controller is then referenced to check whether it can issue the instruction at the current PC and copy it to the instruction queue or not. It checks whether there is a free *Reservataion Station* and that there were no preceeding *JAL/JALR* instructions.

If the controller was able to issue it, then it is transfared to a free reservataion station of the corresponding type. Since this is only a single-issue processor, there can only be one instruction issued at a time, and it will always be the top of the queue instruction.

If the instruction issued is a *JAL/JALR* instruction, a flag is set to not issue any more instructions until the jump instruction is evaluated.

If the instruction issued is a *BEQ* instruction, a flag is set to record all the instructions after the branch to prevent them from executing as long as the branch outcome was not determined.

The reservataion station chosen then marks rd in the register file producing unit with the name of the station, and checks the values of the producing unit of rs1 and rs2 according to each instruction type.

Instruction execution

The controller checks for every busy reservataion station some conditions.

- The instruction doesn't have any dependencies and that both Q_j and Q_k are empty (*Both operands are ready*)
- The instruction is not an instruction after branch instruction.
- The instruction doesn't have a *LOAD-STORE* dependency and can be safely executed.

if all the previous condiitons are true, it then starts executing and marks this cycle as *started-execution-cycle*. If it started executing then, it will never stop until it finishes.

It keeps on executing the instruction until it finishes and if it finishes, it changes its stage to the *Write back* stage and marks the end of executation cycle.

Instruction write-back

Since we have only one common data bus, one instruction can only be written back at a time, even if all instruction did finish executation in their respecctive stations. It also writes back according to the order of issuing, which means if there are two instructions that finished in the same exact cycle, the first one to be issued of the two will always be chosen over the second one.

The only condition that an instruction writes-back is that if there is not **WAW** dependencies.

If it manages to enter the write-back stage, then it writes its value to the common data bus and writes its value to the register file.

If the instruction we are writing back is *JAL/JALR*, we jump to the specified address and start issuing instructions again by setting the flag to false.

If the instruction we are writing back is *BEQ*, we have two options.

- Either the branch is taken:
  All the instructions after the branch instruction is then cleaned and reset their state (Flushed), and it will also be marked as FLUSHED and will be seen in the simulation as so.
  The effective address where we will take the instruction instruction is modified to be the target address.

- Branch is not taken:
  We clean the instructions after the branch and allow them to execute normally.
  We have to make sure that we are not cleaning the instructions after a second branch instruction since those ones do not belong to the one we are writing back right now.

The Common Data Bus

The Common Data Bus (CDB) goes around each busy station and sees if it has the value requested by that station. It then changes it in the station and removes the respective producing unit in the station (Q_j or Q_k).

NOTE: The order which those four stages is done in the simulation is this:

1. Instruction issuing.
2. Instruction write-back.
3. Common Data Bus sweep.
4. Instruction executation

This is because some instructions will require the values coming from the CDB before executing. If we were to arrange them differently this will require it to stall for one cycle before it executes which is not intended.

4 The Green Table

The green table is just a way of writing the cycles of each stage each instruction went through. You can log to a file after you finished executing your program to "Green_Table.txt".

The green table will contain different components:

- The number of cycles you advanced.
- The last cycle an instruction wrote back.
- the IPC, (Instruction Per Cycle)
- Branch Misprediction percentage
- The green table
- The data Memory content

5 Used Hardware

The Operating System used to develop this project is Linux Mint 20.2 Cinnamon.

The PC Specs:

- Processor: Intel© Core™ i7-4810MQ CPU @ 2.80GHz × 4
- Memory: 8Gb
- Harddrive: 750Gb
- Graphics Card: Intel Corporation 4th Gen Core Processor Integrated Graphics Controller.
- External Graphics Card: AMD Radeon HD 8790M

This project was tested and run on both Linux Mint operating system, and Windows 10 Pro. You can find a link to the repositry on github by following this link:
https://github.com/Seif-Sallam/TomasolosAlgorithm

Used Softwares

The main compilar used to compile the source code was *g++* on Linux and *MSVC* on Windows 10 Pro.

Libraries such as SFML, and ImGui were used for the graphical user interface, as well as implementation of different extensions for the ImGui library such as ImGuiFileDialog to open the dialog and ImGui-SFML.

To find SFML you can follow this link for the official website: https://en.sfml-dev.org/index.php

SFML on github: https://github.com/SFML/SFML

ImGui on Github: https://github.com/ocornut/imgui

ImGui-SFML: https://github.com/eliasdaler/imgui-sfml

ImGuiFileDialog: https://github.com/aiekick/ImGuiFileDialog/

6 Compiling the project

Linux

To compile the project on Unix-based systems (Linux tested):

You will need to install SFML dependencies. This is done using this command in any terminal:

```
sudo apt-get install libsfml-dev
```

Then after installing all the dependencies, you will do these following steps:

Open a terminal in the directory you want to put the project in then:

```
1- git clone https://github.com/Seif-Sallam/TomasolosAlgorithm.git
2- cd TomasolosAlgorithm
3- make
4- ./Proj.out
```

The makefile output is *Proj.out*, this is the executable binary file.

Some issues that you may encounter

- If the command mkdir did not create the needed directory:
  Create the directories "imguiObjects" & "Objs" in the same directory as the makefile
- If you run the project and press OK without selecting a file, the program will crash.
  This is something I am not able to prevent since the code base for opening file dialog is not mine and I cannot really change it.

Windows

To get the Windows version of the same project that contains a Visual Studio solution with all the libraries linked and working, you will need to clone the repositry and checkout to the Windows Branch.

Open a terminal in the directory you want to put the project in then:

```
1- git clone https://github.com/Seif-Sallam/TomasolosAlgorithm.git
2- cd TomasolosAlgorithm
3- git checkout Windows
```

You then need to open the visual studio solution (Tested on Visual Studio 2019 and Visual studio 2022).

The supported archticture is x86 with both Release and Debug modes.

All the libraries and dependencies are included in the project and there are nothing to be added or downloaded.

Mac-OS

I have tried using Mac-OS and testing the project on it, but the compiler, *clang*, used there has many differences between it and the ones used in both Windows *MSVC Compiler*, or Linux *g++ and gcc*. The differences in their library implementation and usage created some spontanous errors that could be solved by using a higher version of c++ like *c++17*, yet it never solved the issues with it compiling and running SFML.

I was not able to create a Mac-OS compatible version of the project since I do not own a Mac-OS based system, and was merely able to test the code on the Macbook of one of my colleagues.

7 Test cases

Default Program

The default program that is fed into the program every time the program is opened for the first time implements a simple test case that tests the use of every instruction and every register.

It contains a loop that is executed 5 times and saves several data into the memory at several locations and sometimes loads them. The instructions can be found in the file “DefaultProgram.txt” in the folder Tests.

```
Number of Cycles: 77
Last Instruction Cycle: 73
IPC: 0.520548
Branch misprediction percentage: 20%

Green Table:
```

PC	Instruction	Issue	Execution Start	Executation End	Write back
0	ADDI R1, R0, 10	0	1	2	3
1	ADDI R2, R0, -10	1	2	3	4
2	STORE R1, 0(R0)	2	4	6	7
3	STORE R2, 1(R0)	3	5	7	8
4	LOAD R3, 0(R0)	4	6	8	9
5	ADDI R4, R3, 4	5	9	10	11
6	ADD R4, R4, R2	6	11	12	13
7	STORE R4, 2(R0)	8	10	14	15
8	NEG R5, R2	9	10	12	14
9	STORE R5, 3(R0)	10	12	15	16
10	ABS R6, R2	11	12	13	17
11	STORE R6, 4(R0)	16	18	20	21
12	ADDI R1, R0, 26	17	18	19	20
13	ADDI R2, R0, 2	18	19	20	22
14	DIV R3, R1, R2	19	22	31	32
15	STORE R3, 5(R0)	20	22	33	34
16	ADDI R1, R0, 0	21	22	23	24
17	ADDI R7, R0, 5	22	23	24	25
18	ADD R1, R1, R7	23	25	26	27
19	ADDI R7, R7, -1	25	26	27	28
20	BEQ R7, R0, 1	26	28	29	30
21	JAL R3, -4	27	30	31	33
18	ADD R1, R1, R7	34	35	36	37
19	ADDI R7, R7, -1	35	36	37	38
20	BEQ R7, R0, 1	36	38	39	40
21	JAL R3, -4	37	40	41	42
18	ADD R1, R1, R7	43	44	45	46
19	ADDI R7, R7, -1	44	45	46	47
20	BEQ R7, R0, 1	45	47	48	49
21	JAL R3, -4	46	49	50	51
18	ADD R1, R1, R7	52	53	54	55
19	ADDI R7, R7, -1	53	54	55	56
20	BEQ R7, R0, 1	54	56	57	58
21	JAL R3, -4	55	58	59	60
18	ADD R1, R1, R7	61	62	63	64
19	ADDI R7, R7, -1	62	63	64	65
20	BEQ R7, R0, 1	63	65	66	67
21	JAL R3, -4	FLUSHED	FLUSHED	FLUSHED	FLUSHED
22	STORE R1, 6(R0)	68	70	72	73

The number of cycles given for each instruction is the default as the ones given in the project description.

This program shows several aspects of the schedualing of this algorithm:

- Instructions can be executed out of order:
  We can see this in instruction at PC 9 and 8, where the *STORE* instruction started execting before the *ADD* instruction.

- Instruction can also write back out of order:
  We can see this in instruction at PC 16 and 15, where instruction at 15, (*DIV*) finished at cycle 34 while the instruction at PC 16, (*ADDI*) finished at cycle 24.

- Instructions do not execute before the operands are ready:
  In instruction at PC 5 and 6, since instruction 6 needs the operanrd rs1 which is produced by instruction at PC 5.

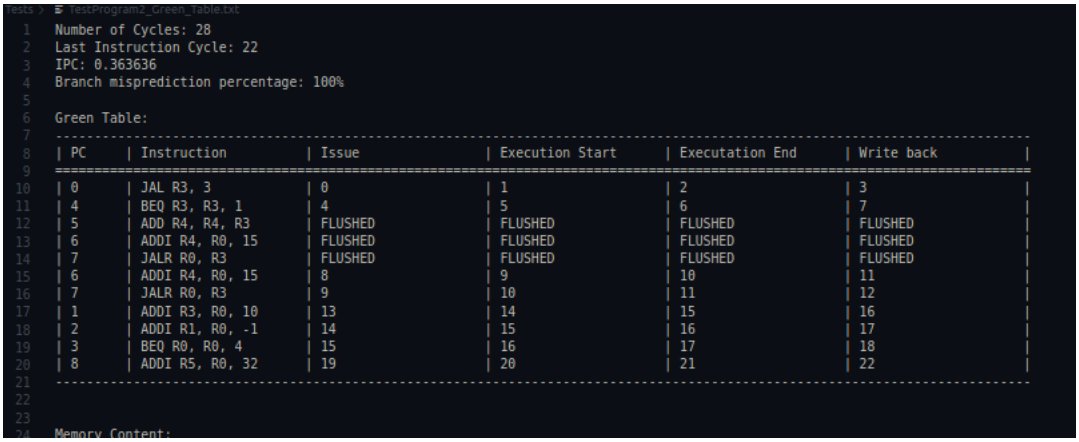
- What is after JAL/JALR instructions do not get issued unless JAL/JALR instructions are done.
  We can see this in the program at PC 21, where the JAL instruction finished at cycle 33 and the one after it *ADD* was issued after *JAL* finished writing back, even though there is a free reservataion station.

- If the branch is taken, all the instructions after the branch instruction was flushed.
  We can see this in the last two instructions of the program.

- The memory content is updated as you can see in the green table for that program.

Test Program 2

Test Progam 2 is designed to test JAL and JALR instructions specifically, by creating some instructions that never enter the queue as they were jumped from. There are also some branch isntructions in the middle to make sure we do not get in an infinite loop. The program jumps back and fourth into differnet instructions.



Tests > TestProgram2_Green_Table.txt

```
Number of Cycles: 28
Last Instruction Cycle: 22
IPC: 0.363636
Branch misprediction percentage: 100%

Green Table:
```

PC	Instruction	Issue	Execution Start	Executation End	Write back
0	JAL R3, 3	0	1	2	3
4	BEQ R3, R3, 1	4	5	6	7
5	ADD R4, R4, R3	FLUSHED	FLUSHED	FLUSHED	FLUSHED
6	ADDI R4, R0, 15	FLUSHED	FLUSHED	FLUSHED	FLUSHED
7	JALR R0, R3	FLUSHED	FLUSHED	FLUSHED	FLUSHED
6	ADDI R4, R0, 15	8	9	10	11
7	JALR R0, R3	9	10	11	12
1	ADDI R3, R0, 10	13	14	15	16
2	ADDI R1, R0, -1	14	15	16	17
3	BEQ R0, R0, 4	15	16	17	18
8	ADDI R5, R0, 32	19	20	21	22

```


Memory Content:
```

The number of cycle given for each instruction is the default as the ones given in the project description This program shows the flushing of the instructions as well as the jump over the instructions and that they do not even get issued.

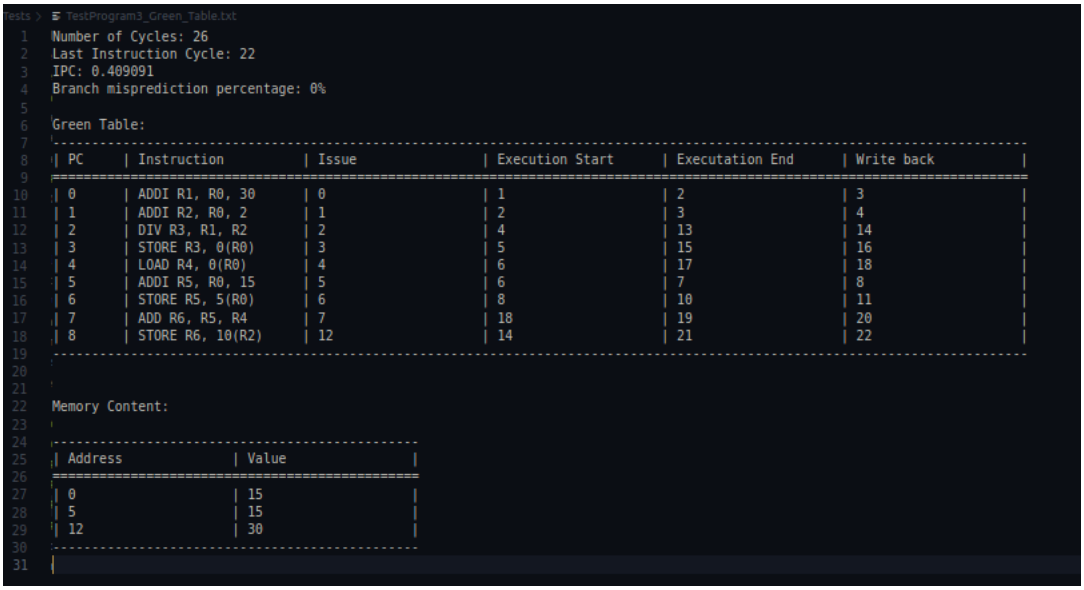
An example of that is the instruction at PC 5. It is flushed and never executed since there is a branch instruction that is taken and doesn't let it execute.

The branch misprediction percentage in this example is 100% because all the branch instructions in this program are taken.

Test Program 3

Test Program 3 is designed to test LOAD-STORE hazards and to check whether a load instruction will wait until the preceeding store instruction finishes or not.

One thing to notice is that the laod and store instructions mark starting execution by the end of computing the memory address regardless of having any dependencies or not. However, it will only continue executing if and only if there exists no dependencies



Tests > TestProgram3_Green_Table.txt

```
Number of Cycles: 26
Last Instruction Cycle: 22
IPC: 0.409091
Branch misprediction percentage: 0%

Green Table:

| PC     | Instruction          | Issue              | Execution Start     | Executation End     | Write back          |
|========|======================|====================|=====================|=====================|=====================|
| 0      | ADDI R1, R0, 30      | 0                  | 1                   | 2                   | 3                   |
| 1      | ADDI R2, R0, 2       | 1                  | 2                   | 3                   | 4                   |
| 2      | DIV R3, R1, R2       | 2                  | 4                   | 13                  | 14                  |
| 3      | STORE R3, 0(R0)      | 3                  | 5                   | 15                  | 16                  |
| 4      | LOAD R4, 0(R0)       | 4                  | 6                   | 17                  | 18                  |
| 5      | ADDI R5, R0, 15      | 5                  | 6                   | 7                   | 8                   |
| 6      | STORE R5, 5(R0)      | 6                  | 8                   | 10                  | 11                  |
| 7      | ADD R6, R5, R4       | 7                  | 18                  | 19                  | 20                  |
| 8      | STORE R6, 10(R2)     | 12                 | 14                  | 21                  | 22                  |


Memory Content:

| Address              | Value              |
|======================|====================|
| 0                    | 15                 |
| 5                    | 15                 |
| 12                   | 30                 |
```

We can see how the load instruction at PC 4 starts fetching from memory after the store instruction before it finishes writing back.

Some instructions finish writing back before the others because of the lack of the WAW dependencies.

Licenses

All the used material in this project is under MIT license.