University of California, Riverside
Department of Computer Science & Engineering

Title: Advanced Computer Architecture Final Project Report

Student Name:
Mahbod Afarin

Student ID:
862186340

Winter 2020

Table of Contents

1. Introduction

In this project, I implemented Tomasulo Speculative algorithm with C++. This simulator takes assembly instructions in a text file and then it decoded the instructions to drive the Tomasulo Speculative simulator. It also takes a configuration text file to setup some parameter for the algorithm. we can specify the number of reservation stations for each execution units, execution time for each execution units, the number of memory latency cycles for load and store instructions, the number of execution units, and the number of ROB entries. Also, we can initialize the memory and registers using our configuration file. The output of the simulator is a table which specify issue, execution, memory and write-back cycle for each instruction.

2. Implementation

In this section we want to have a quick overview on the speculative Tomasulo architecture and then we will discuss about the implementation. For this purpose, we should first explore the architecture of a simple Tomasulo algorithm. Therefore, in the first section, I will explore the simple Tomasulo architecture, and then in the second section, I will talk about the speculative Tomasulo architecture. In the final section, I will talk about the implementation of the algorithm.

2.1 Tomasulo architecture

Tomasulo algorithm is a hardware algorithm for dynamic scheduling of instructions that allows out of order execution and enables more efficient of multiple execution units. In this algorithm, the issue stage will be done in order but execution and write back will be done out of order. In the figure 1, we have shown the details of the Tomasulo architecture.

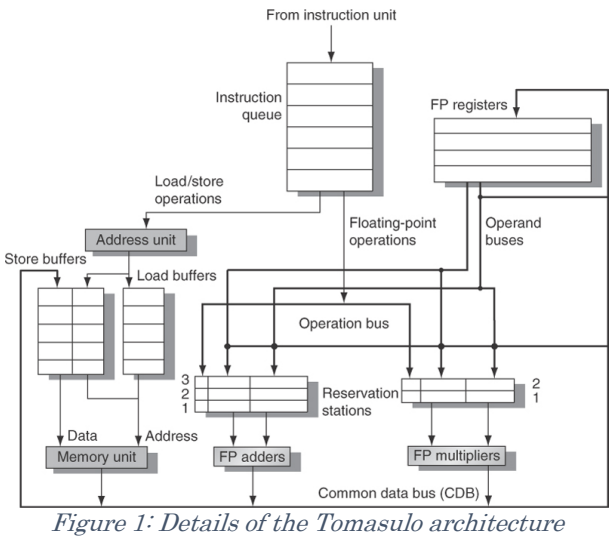

Figure 1: Details of the Tomasulo architecture

Tomasulo Algorithm uses register renaming to correctly perform out-of-order execution. All general-purpose and reservation station registers hold either a real value or a placeholder value. If a real value is unavailable to a destination register during the issue stage, a placeholder value is initially used. The placeholder value is a tag indicating which reservation station will produce the real value. When the unit finishes and broadcasts the result on the common data bus, the placeholder will be replaced with the real value.

2.2 Speculative Tomasulo architecture

One of the major problems with Tomasulo algorithm is that when we faced branch instruction it will not issue the instructions after the branch because it dose not know whether that branch is taken or not taken. Therefore, it stalls all the instructions after the branch to finds out the results if the branch instruction. These stalls lead to the wasting time on CPU. For solving this problem, we can use speculative Tomasulo. As we have shown in the figure 2, speculative Tomasulo uses the Re-order Buffer (ROB) to issue the instructions after the branch instruction. We will add the commit stage to the speculative Tomasulo pipeline and in this stage, we write the results in the ROB instead of register file or memory. So, when the instruction is no longer speculative, the speculative Tomasulo allow it to update the register file or memory.

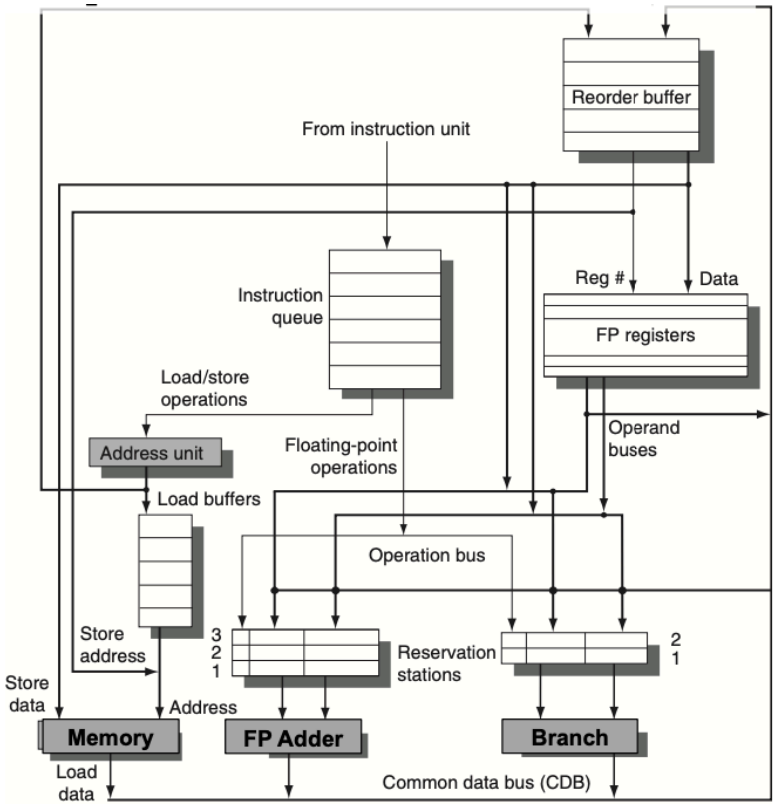

Figure 2: Speculative Tomasulo Architecture

The commit stage will allow us to complete the instructions in-order and in this way, we can execute the instructions after the branch. So, as we have shown in the table 1, the main deference between simple Tomasulo and speculative Tomasulo is that in simple Tomasulo we complete the instruction out-of-order, but in speculative Tomasulo we complete the instructions in-order. In this project I implemented the speculative Tomasulo.

Table 1: The main deference between Tomasulo and speculative Tomasulo

Operation	Issue	Execution	Completion
Tomasulo	In-order	Out-of-order	Out-of-order
Speculative Tomasulo	In-order	Out-of-order	In-order

2.3 Implementation of the speculative Tomasulo

Speculative Tomasulo consist of 5 main stages. These five stages are issue, execution, memory, write-back, and commit. In the issue stage, the simulator fetches the instructions from the instruction queue and then put the instructions to the reserve stations. In the execution stage the simulator gets the instructions from reservation stations and then executes the instructions. The memory stage will use only for load and store instructions. If we have load instruction, the simulator load the data from memory and then it will broadcast the data using common data bus, but for the store instruction, it will write the result in the Rob and when the instruction has committed, it will store the data in the memory. In the write-back stage, the simulator writes the results into Rob and also broad cast the results for other instructions. In the commit stage, the simulator writes the results to the memory and register files and then it will delete the instructions in the reserve stations.

For the branch instructions, it assumes that the branch is taken and then it will issue the instruction which is the destination of the branch. If the prediction is wrong, it will flush the wrong instructions and then it will issue the normal instructions after the branch instruction. The simulator has 16 integer registers and 16 floating point registers.

This simulator has two inputs, the first input is the assembly input instructions, and the second is the configuration text file for the speculative Tomasulo architecture. The instruction set of the simulator is in the table 2.

Table 2: The instructions which supported by simulator

Instruction	Explanation
Add	Integer Adder
Sub	Integer Subtractor
Addf	Floating Point Adder
Subf	Floating Point Subtractor
Ld	Load from Memory
Sd	Store to Memory
Mulf	Floating Point/Integer Multiplier
Bne	Branch not Equal to Zero
Beq	Branch Equal to Zero

In the table 3, I show the items which the user can change for the speculative Tomasulo architecture.

Table 3: The configuration of the simulator

Configurable items of the speculative Tomasulo simulator
Number of the reserve stations
Execution cycles of each execution units
Memory latency for load/store instruction
Number of the execution units
Initial value for registers
Initial values for memory
Number of the Rob entries

The speculative Tomasulo simulator stores the results of the simulation in the output text file. The output of the similar is in the table format like we had in the lectures of the course. It also contains the register file values and the memory values after executing the simulator. In the next section we will see the sample output of the simulator.

3. Execution and timing results

For the running the simulator first we should configure the architecture of our simulator in the *configuration.txt*. In this text file we can specify the number of reservation stations for each execution units, execution cycles for each execution units, memory latency cycles for load and store instructions, the number of execution units, and the number of ROB entries. Also, we can initialize the memory and registers using our configuration file. This simulator will take assembly instructions using *input-instructions.txt*. So, we can specify our input instruction on that text file. For running the simulator we should first *make* it and the using *./tomasulo-simulator* command.

3.1 Running sample instruction1

In this section, I am going to run the simulator with a sample instruction input to test the program. I have shown the sample input instructions in the table 4.

Table 4: Sample input test 1

1	Ld R4, 0(R1)
2	Ld R5, 0(R2)
3	Add R5, R4, R5
4	Sd R5, 0(R1)
5	Ld R6, 0(R1)
6	Mulf R3, R6, R3
7	Sub R3, R3, 53
8	Sd R3, 0(R1)

In the configuration input text, I set 3 reserve station for integer adder, 5 reserve station for load, and 2 reserve station for multiplier. The execution cycles for integer adder, load, store instructions are 1, the execution cycles for multiply is 15, and the memory cycles for load and store is 3. We have 1 execution unit per instruction and the number of Rob entries is 9. In addition, I set the initial value of $R1 = 100$, $R2 = 108$, $R3 = 10$, $Mem[100] = 5$, and $Mem[108] = 10$. In the figure 3, we can see the results of the execution sample instruction1.

	ISSUE	EXECUTION	MEMORY	WRITE-BACK	COMMIT
Ld R4, 0(R1)	1	2	3	6	7
Ld R5, 0(R2)	2	3	6	9	10
Add R5, R4,R5	3	9		10	11
Sd R5, 0(R1)	4	10	11	14	15
Ld R6, 0(R1)	5	11	14	17	18
Mulf R3, R6, R3	6	17		32	33
Sub R3, R3, 53	7	32		33	34
Sd R3, 0(R1)	8	33	34	37	38

Figure 3: The result of the execution of the test 1

I have shown the output of the register file and the memory in the figure 4. As we can see in the figure 4, the content of the $Mem[100] = 97$ which shows that the result of the simulation is correct.

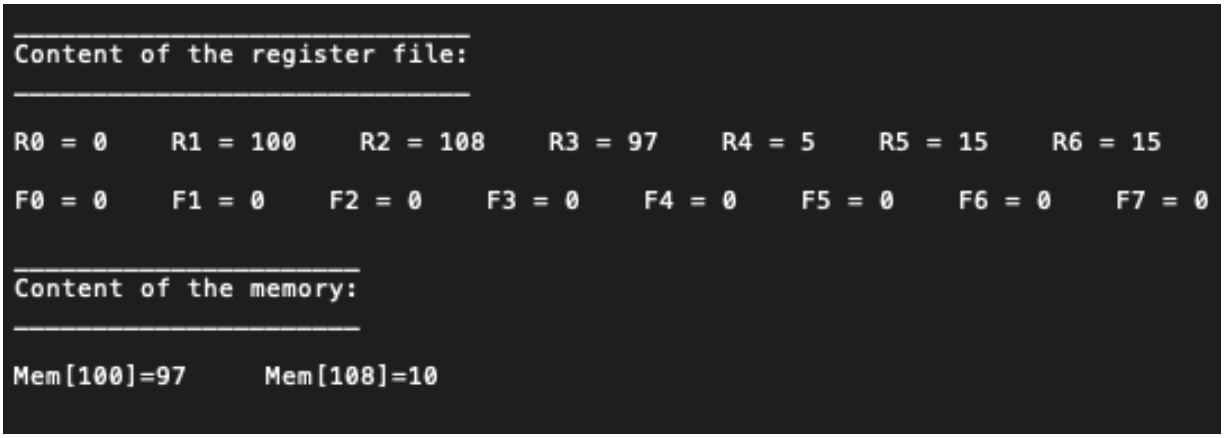

Figure 4: The content of the register file and memory for test 1

3.2 Running sample instruction 2

In this section, I am going to run the simulator with a sample instruction input which contain branch instruction to test the branch instruction. I have shown the sample input instructions in the table 5.

Table 5: Sample input test 2

1	Ld R2, 0(R1)
2	Add R2, R2, 1
3	Sd R2, 0(R1)
4	Bne R2, R3, -2

In the configuration input text, I set 3 reserve station for integer adder and 5 reserve station for load. The execution cycles for integer adder, load, store, and branch instruction is 1 and the memory cycles for load and store is 3. We have 1 execution unit per instruction and the number of Rob entries is 9. In addition, I set the initial value of $R1 = 100$, $R3 = 10$, and $Mem[100] = 5$. The result of the execution is in the figure 5.

	ISSUE	EXECUTION	MEMORY	WRITE-BACK	COMMIT
Ld R2, 0(R1)	1	2	3	6	7
Add R2, R2,1	2	6		7	8
Sd R2, 0(R1)	3	7	8	11	12
Bne R2, R3,-2	4	7		8	13
Add R2, R2,1	7	8		9	14
Sd R2, 0(R1)	8	9	11	14	15
Bne R2, R3,-2	9	10		12	16
Add R2, R2,1	10	12		13	17
Sd R2, 0(R1)	11	13	14	17	18
Bne R2, R3,-2	12	13		15	19
Add R2, R2,1	13	15		16	20
Sd R2, 0(R1)	14	16	17	20	21
Bne R2, R3,-2	15	16		18	22
Add R2, R2,1	16	18		19	23
Sd R2, 0(R1)	17	19	20	23	24
Bne R2, R3,-2	18	19		21	25

Figure 5: The result of the execution of test 2

The output of the registers is in the figure 6. As we can see, the final value of the R2 register is 10 and this will the end of our loop.

```
_____________________________
Content of the register file:
_____________________________

R0 = 0    R1 = 100    R2 = 10    R3 = 10

F0 = 0    F1 = 0    F2 = 0    F3 = 0

_____________________
Content of the memory:
_____________________

Mem[100]=10
```

Figure 6: The output of the registers for test 2

3.3 Running sample instruction 3

In the table 6, I show the sample test instruction 3 for testing the simulator. In the configuration input text, I set 3 reserve station for integer adder, 3 reserve station for floating point adder, 5 reserve station for load, and 2 reserve station for multiplier. The execution cycles for integer adder, load, store instructions are 1, the execution cycle for multiplier is 15, the execution cycle for floating point adder is 5, and the memory cycles for load and store is 3. We have 1 execution unit per instruction and the number of Rob entries is 9. In addition, I set the initial value of $R1 = 100$, $R3 = 5$, $R4 = 111$, $R7 = 115$, $R10 = 0$, $Mem[100] = 5$, $Mem[111] = 2$, and $Mem[115] = 2$.

Table 6: Sample test instruction 3

1	Ld R2, 0(R1)
2	Add R2, R2, 1
3	Sd R2, 0(R1)
4	Bne R2, R3, -2
5	Ld F2, 0(R4)
6	Mulf F2, F2, 5
7	Subf F2, F2, 10
8	Ld R8, 0(R7)
9	Sub R8, R8, 1
10	Bne R8, R10, -1

The result of the execution is in the next page in figure 7.

	ISSUE	EXECUTION	MEMORY	WRITE-BACK	COMMIT
Ld R2, 0(R1)	1	2	3	6	7
Add R2, R2, 1	2	6		7	8
Sd R2, 0(R1)	3	7	8	11	12
Bne R2, R3, -2	4	7		8	13
Add R2, R2, 1	7	8		9	14
Sd R2, 0(R1)	8	9	11	14	15
Bne R2, R3, -2	9	10		12	16
Add R2, R2, 1	10	12		13	17
Sd R2, 0(R1)	11	13	14	17	18
Bne R2, R3, -2	12	13		15	19
Add R2, R2, 1	13	15		16	20
Sd R2, 0(R1)	14	16	17	20	21
Bne R2, R3, -2	15	16		18	22
Add R2, R2, 1	16	18		19	23
Sd R2, 0(R1)	17	19	20	23	24
Bne R2, R3, -2	18	19		21	25
Add R2, R2, 1	19	21		22	26
Sd R2, 0(R1)	20	22	23	26	27
Bne R2, R3, -2	21	22		24	28
Add R2, R2, 1	22	24		25	29
Sd R2, 0(R1)	23	25	26	29	30
Bne R2, R3, -2	24	25		27	31
Add R2, R2, 1	25	27		28	32
Sd R2, 0(R1)	26	28	29	32	33
Bne R2, R3, -2	27	28		30	34
Add R2, R2, 1	28	30		31	35
Sd R2, 0(R1)	29	31	32	35	36
Bne R2, R3, -2	30	31		33	37
Add R2, R2, 1	31	33		34	38
Sd R2, 0(R1)	32	34	35	38	39
Bne R2, R3, -2	33	34		36	40
Ld F2, 0(R4)	34	35	38	41	42
Mulf F2, F2, 5	35	41		56	57
Subf F2, F2, 10	36	56		61	62
Ld R8, 0(R7)	37	38	41	44	63
Sub R8, R8, 1	38	44		45	64
Bne R8, R10, -1	39	45		46	65
Sub R8, R8, 1	45	46		47	66
Bne R8, R10, -1	46	47		48	67

Figure 7: Result of the execution sample 3