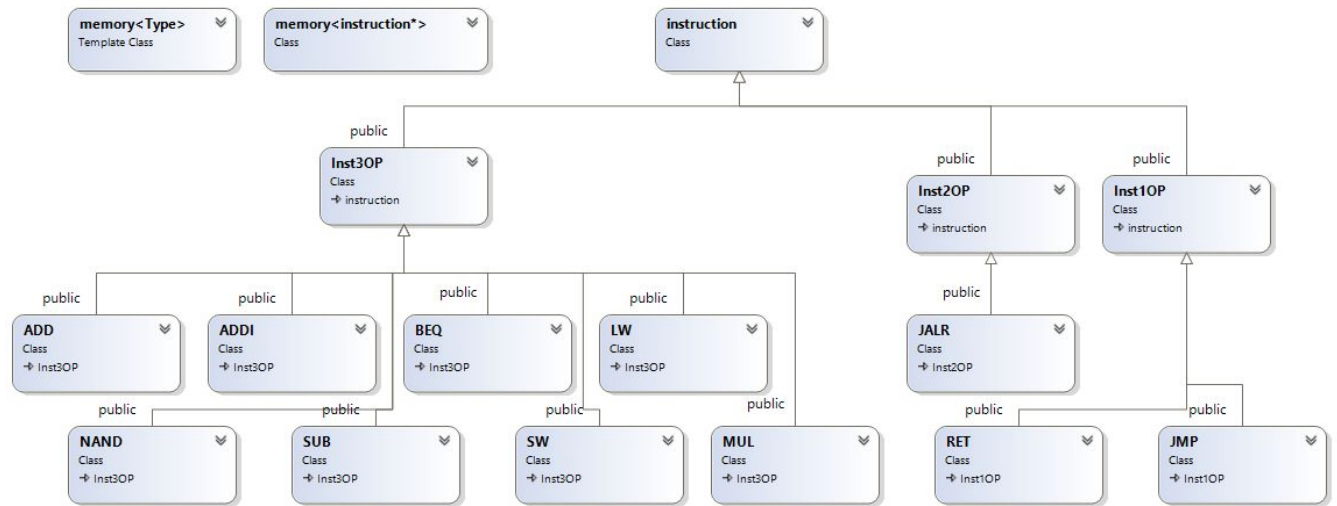


# Project 2: Tomasulo's Algorithm

## Simulation

### Final Milestone



Laila N ElKoussy  
900160812  
Yahia Khaled Farid  
900161331  
Yousef Mohab Koura  
900160083

# Table of Contents

<b>Introduction</b>	<b>2</b>
<b>Our Simulator</b>	<b>2</b>
Interface	2
Instruction Set Architecture and Format	3
Code and Implementations	4
Main Function	4
Simulator Class	4
Memory Class	5
Instruction Class	6
<b>User Guide</b>	<b>6</b>
The Code	6
Cycle 1	7
Cycle 2	7
Cycle 3	9
Cycle 4	11
Cycle 5	12
Cycle 6	13
Cycle 7	13
Cycle 8	14
Cycle 9	15
Cycles 10 to 12	15
Cycle 13	15
Cycle 14	15
<b>Results</b>	<b>16</b>
<b>Appendix</b>	<b>17</b>

# Introduction

The Central Processing Unit (CPU) is the heart of any electronic device. They vary in types, functions, and design, but their efficiency mainly lies in how many instructions they can execute in a certain given time. The most popular kind of design for a CPU is a pipelined CPU. Unfortunately, some instructions, like floating point instructions, can lag the whole pipeline, and therefore hinder the performance. There is some approach that allows us to limit the effect of that; which is having a superscalar out-of-order CPU. In this project, we will be simulating simplified superscalar out-of-order 16-bit RISC processor that uses Tomasulo's algorithm with speculation.

## Our Simulator

In this milestone, our simulator does not implement Tomasulo's algorithm. It only takes the input and parses the content in the corresponding containers. Our simulator follows a Harvard architecture, which means it has separate 128 KB memories for the data and the instructions.

## Interface

Our interface first prompts the user to enter the starting address in hexadecimal. After this, it asks the user to choose between two inputting modes.

The first mode is for the user to enter instruction by instruction manually. The second is to read directly from a text file. After taking the input, the simulator echoes it so that the user can make sure that the instructions entered are correct. If taking input from a file, the file needs to specify the .text and .data sections. It is allowed to have multiple .text and .data sections, but it is imperative that the indicators be there.

## Instruction Set Architecture and Format

- Load word: Loads value from memory into regA. Memory address is formed by adding imm with contents of regB, where imm is a 7-bit signed immediate value (ranging from -64 to 63).

`LW regA, regB, imm`

- Store word: Stores value from regA into memory. Memory address is computed as in the case of the load word instruction.

`SW regA, regB, imm2.`

- Jump: branches to the address PC+1+imm

`JMP imm`

- Branch if equal: branches to the address PC+1+imm if regA=regB

`BEQ regA, regB, imm`

- Jump and link register: Stores the value of PC+1 in regA and branches (unconditionally) to the address in regB.

`JALR regA, regB`

- Return: branches (unconditionally) to the address stored in regA

`RET regA`

- Add: Adds the value of regB and regC storing the result in regA

`ADD regA, regB, regC`

- Subtract: Subtracts the value of regC from regB storing the result in regA

`SUB regA, regB, regC`

- Add immediate: Adds the value of regB to imm storing the result in regA

`ADDI regA, regB, imm`

- Nand: Performs a bitwise NAND operation between the values of regB and regC storing the result in regA

`NAND regA, regB, regC`

- Multiply: Multiplies the value of regB and regC storing the result in regA

`MUL regA, regB, regC`

## Code and Implementations

Our code so far only takes input from the user and places it in the instruction memory and data memory. But, we have a whole skeleton ready for the second milestone. Since we have multiple classes and an inheritance tree, we have appended our class diagram (appendix 1).

### Main Function

Our main function is the one in charge of the user interface. Based on user input, the starting address is chosen, the file name is entered, or the instructions are entered one by one.

### Simulator Class

The simulator is in its own class (`SIM.h`). It's the one containing all of the memories, the program counter, and it is the one mediating between the instructions and handling the flow of the program. It has the five stages of the pipeline; fetch, issue, execute, writeback, and commit. The stages' order is inverted so an instruction doesn't fully execute in one clock cycle, so we have commit, execute and writeback, issue and fetch. We will be discussing each stage thoroughly.

#### 1. Fetch

In this stage, we start by checking for the capacity of the instruction queue. If the queue is not full, we push an instruction from the instruction memory into the instruction queue. Then, we recheck for the instruction queue capacity. If it isn't full, we push another instruction into the queue.

#### 2. Issue

In the issue stage, we check if the instruction queue is empty. If it isn't empty, we then check if the top of the queue is a load/store instructions. If it is, we check that the register it is loading or storing to/from are valid. If it isn't valid, we stall. If it isn't a load/store instructions, we check if it is an unconditional jump instruction (jalr or ret). If it is an unconditional jump, we stall in order to be able to jump to the correct order and not fetch anymore instructions from the wrong address. If it is neither a load/store instruction or an unconditional jump, we check for functional unit hazards, and if there are none, we pop the instruction at the top of the instruction queue and then issue it. After issuing the first instruction, we recheck the instruction queue to see if it has instructions or not. If it's not empty, we recheck the previously mentioned cases for the instruction currently at the

front of the queue. If it passes everything without requiring stalling, we then check if there is data dependency between the two instructions we want to issue. If there are none, we issue both instructions, but if there are we only issue the first instruction.

### 3. Execute

In the execute stage, we are just iterating over all of the functional units and the instructions running in them and calling each instruction's respective execute function. The execute function returns a bool that indicates when the instruction is done executing (this depends on how many clock cycles have passed since the instruction has been committed). When that bool is true, this instruction is set to stop executing.

### 4. Writeback

In the writeback stage of the instructions, we still loop over the functional unit and the instructions running in them, except that now we are checking if the instruction is done executing. If it is done executing, then we call its writeback function, which signals it as ready in the reorder buffer, and empty out its place in its corresponding functional unit.

### 5. Commit

In the commit stage, we are mainly just checking our reorder buffer. If the instruction at the front of the reorder buffer is ready (i.e. has been executed and has the result available), we check if it's a branch instruction. If it isn't a branch instruction, we just commit the changes. If it is a branch instruction, we check our prediction of it. If it's a misprediction, we commit the branch instruction then we empty the instruction queue and clear out all of the instructions being processed in the functional units.

At the end of the simulate function, after all the instructions are committed, the relevant statistics are displayed (IPC, branch prediction accuracy, cycles elapsed).

## Memory Class

Our memories are 128KB, they are word-addressed so they're indexed using 64KB, instead of 128KB. Our memory class is a template class (`Memory.h`), we use it for both the instruction and data memory.

## Instruction Class

In order to be able to have one holder for all types of instructions, we have created an instruction class (`Instruction.h`) that serves as a base-class for our different types of instructions. Then, we have three classes inheriting from it, `Inst1OP`, `Inst2OP`, `Inst3OP`, each with a header file using the same names. These three classes are designed to separate between the number of operands in the instructions. After this, we have a dedicated class for each instruction, each inheriting from its corresponding operand class.

## User Guide

In this section, we will be giving the user a step-by-step guide of how to run our simulator, with explanation of what is happening in each clock cycle.

## The Code

```
.text

ADDI x2,x0,5

ADDI x3,x0,10

LW x1,x0,10

SUB x4,x3,x2

MUL x5,x2,x3

NAND x6,x4,x2


.data

0x00:    20

0x01:    45

0x0A:    22
```

## Cycle 1

### Instruction Queue

instq	{ size=0x00000002 }
c	{ size=0x00000002 }
[allocator]	allocator
[0]	0x007f0e48 {parameter1=0x00000000 immediate=0x00000000 p=0x00000000 <NULL> ...}
[ADDI]	{parameter1=0x00000000 immediate=0x00000000 p=0x00000000 <NULL> ...}
__vfptr	0x00f1389c {Architecture Project 2.exe!const ADDI::vftable'} {0x00ee6031 {Architecture Project 2.exe!Inst3OP: ...}}
instruction_name	"ADDI"
whole_instruction	"ADDI x2,x0,5"
operand1	0x00000002
ID	0x00000000
operand2	0x00000000
operand3	0x00000005
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"ADD"
[1]	0x007f1700 {parameter1=0x00000000 immediate=0x00000000 p=0x00000000 <NULL> ...}
[ADDI]	{parameter1=0x00000000 immediate=0x00000000 p=0x00000000 <NULL> ...}
__vfptr	0x00f1389c {Architecture Project 2.exe!const ADDI::vftable'} {0x00ee6031 {Architecture Project 2.exe!Inst3OP: ...}}
instruction_name	"ADDI"
whole_instruction	"ADDI x3,x0,10"
operand1	0x00000003
ID	0x00000001
operand2	0x00000000
operand3	0x0000000a
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"ADD"

I0 and I1 were fetched from the memory and placed in the instruction queue. These two instructions are ADDI x2,x0,5 and ADDI x3,x0,10. The ROB and the stations are currently empty.

## Cycle 2

I2 and I3 are now fetched and placed into the instruction queue. These are LW x1,x0,10 and SUB x4,x3,x2. I0 and I1 were issued and therefore were removed from the instruction queue. They were also placed into the ROB and the ADD stations.



## Instruction queue

instq	{ size=0x00000002 }
c	{ size=0x00000002 }
[allocator]	allocator
[0]	0x00d817b0 {parameter1=0x00000000 address=0x00000000 immediate=0x00000000 ...}
[LW]	{parameter1=0x00000000 address=0x00000000 immediate=0x00000000 ...}
__vfptr	0x00f13a34 (Architecture Project 2.exe!const LW::'vftable') {0x00ee6031 (Architecture Project 2.exe!Inst3OP::s
instruction_name	"LW"
whole_instruction	"LW x1,x0,10"
operand1	0x00000001
ID	0x00000002
operand2	0x00000000
operand3	0x0000000a
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"LW"
[1]	0x00d81868 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00000000 <NULL> ...}
[SUB]	{parameter1=0x00000000 parameter2=0x00000000 p1=0x00000000 <NULL> ...}
__vfptr	0x00f13b44 (Architecture Project 2.exe!const SUB::'vftable') {0x00ee6031 (Architecture Project 2.exe!Inst3OP::s
instruction_name	"SUB"
whole_instruction	"SUB x4,x3,x2"
operand1	0x00000004
ID	0x00000003
operand2	0x00000003
operand3	0x00000002
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"ADD"

## ROB

ROB	{ size=0x00000002 }
c	{ size=0x00000002 }
[allocator]	allocator
[0]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
[ADDI]	{parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
__vfptr	0x00f1389c (Architecture Project 2.exe!const ADDI::'vftable') {0x00ee6031 (Architecture Project 2.exe!Inst3OP::s
instruction_name	"ADDI"
whole_instruction	"ADDI x2,x0,5"
operand1	0x00000002
ID	0x00000000
operand2	0x00000000
operand3	0x00000005
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"ADD"
[1]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
[ADDI]	{parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
__vfptr	0x00f1389c (Architecture Project 2.exe!const ADDI::'vftable') {0x00ee6031 (Architecture Project 2.exe!Inst3OP::s
instruction_name	"ADDI"
whole_instruction	"ADDI x3,x0,10"
operand1	0x00000003
ID	0x00000001
operand2	0x00000000
operand3	0x0000000a
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"ADD"

## ADD stations

ADD_stations	0x00c6f6d8 {0x00d80e48 {instruction_name="ADDI" whole_instruction="ADDI x2,x0,5" operand1=0x00000000;
[0x00000000]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
[ADDI]	{parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
_vfpvr	0x00f1389c {Architecture Project 2.exe\const ADDI::vftable'} {0x00ee6031 {Architecture Project 2.exe\Inst3OP:
instruction_name	"ADDI" Q
whole_instruction	"ADDI x2,x0,5" Q
operand1	0x00000002
ID	0x00000000
operand2	0x00000000
operand3	0x00000005
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"ADD" Q
[0x00000001]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
[ADDI]	{parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
_vfpvr	0x00f1389c {Architecture Project 2.exe\const ADDI::vftable'} {0x00ee6031 {Architecture Project 2.exe\Inst3OP:
instruction_name	"ADDI" Q
whole_instruction	"ADDI x3,x0,10" Q
operand1	0x00000003
ID	0x00000001
operand2	0x00000000
operand3	0x0000000a
cycles	0x00000003
ready	false
started_ex	false
funcUnit	"ADD" Q
[0x00000002]	0x00000000 <NULL>

## RAT

RAT	0x00c8f690 {0x00000000 <NULL>, 0x00000000 <NULL>, 0x00d80e48 {instruction_name="ADDI" whole_instru
[0x00000000]	0x00000000 <NULL>
[0x00000001]	0x00000000 <NULL>
[0x00000002]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
[0x00000003]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
[0x00000004]	0x00000000 <NULL>
[0x00000005]	0x00000000 <NULL>
[0x00000006]	0x00000000 <NULL>
[0x00000007]	0x00000000 <NULL>

Registers x2 and x3 have been filled in the RAT (Register Alias Table) since these are the destination registers of the first two instructions.

## Cycle 3

In this cycle, I0 and I1 start executing. I2 and I3 are issued while I4 and I5 are fetched.

## Instruction queue

instq	{ size=0x00000002 }
c	{ size=0x00000002 }
[allocator]	allocator
[0]	0x00d81920 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00000000 <NULL> ...}
[MUL]	{parameter1=0x00000000 parameter2=0x00000000 p1=0x00000000 <NULL> ...}
_vfp_ptr	0x00f13a78 {Architecture Project 2.exe!const MUL::vftable'} {0x00ee6031 {Architecture Project 2.exe!Inst3OP: ...}}
instruction_name	"MUL"
whole_instruction	"MUL x5,x2,x3"
operand1	0x00000005
ID	0x00000004
operand2	0x00000002
operand3	0x00000003
cycles	0x00000009
ready	false
started_ex	false
funcUnit	"MUL"
[1]	0x00d819d8 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00000000 <NULL> ...}
[NAND]	{parameter1=0x00000000 parameter2=0x00000000 p1=0x00000000 <NULL> ...}
_vfp_ptr	0x00f13abc {Architecture Project 2.exe!const NAND::vftable'} {0x00ee6031 {Architecture Project 2.exe!Inst3OP: ...}}
instruction_name	"NAND"
whole_instruction	"NAND x6,x4,x2"
operand1	0x00000006
ID	0x00000005
operand2	0x00000004
operand3	0x00000002
cycles	0x00000002
ready	false
started_ex	false

## ROB

ROB	{ size=0x00000004 }
c	{ size=0x00000004 }
[allocator]	allocator
[0]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
[1]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
[2]	0x00d817b0 {parameter1=0x00000000 address=0x0000000a immediate=0x0000000a ...}
[3]	0x00d81868 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81700 {parameter1=0x00000000 imm ...}}
[Raw View]	{...}
[Raw View]	{c={ size=0x00000004 } }

I2 and I3 were added to the ROB when they were issued, meaning the ROB now has 4 entries.

## LW station

LW_stations	0x59ccf6ec {0x00d817b0 {instruction_name="LW" whole_instruction="LW x1,x0,10" operand1=0x00000001 ...}}
[0x00000000]	0x00d817b0 {parameter1=0x00000000 address=0x0000000a immediate=0x0000000a ...}
[0x00000001]	0x00000000 <NULL>

## ADD stations

ADD_stations	0x00ccf6d8 {0x00d80e48 {instruction_name="ADDI" whole_instruction="ADDI x2,x0,5" operand1=0x00000000, operand2=0x00000005}}
▷ [0x00000000]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
▷ [0x00000001]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
▷ [0x00000002]	0x00d81868 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81700 {parameter1=0x00000000 immediate=0x0000000a}}

Note that the three add stations are now occupied. This is because I0 and I1 are being executed, while the I3 was just added this cycle.

## RAT

RAT	0x00c8f690 {0x00000000 <NULL>, 0x00d817b0 {instruction_name="LW" whole_instruction="LW x1,x0,10" operand1=0x00000000, operand2=0x0000000a}}
▷ [0x00000000]	0x00000000 <NULL>
▷ [0x00000001]	0x00d817b0 {parameter1=0x00000000 address=0x0000000a immediate=0x0000000a ...}
▷ [0x00000002]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
▷ [0x00000003]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
▷ [0x00000004]	0x00d81868 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81700 {parameter1=0x00000000 immediate=0x0000000a}}
▷ [0x00000005]	0x00000000 <NULL>
▷ [0x00000006]	0x00000000 <NULL>
▷ [0x00000007]	0x00000000 <NULL>

Registers x1 and x4 have now been filled, since these are the destination registers for I2 and I3.

## Cycle 4

During this cycle, I0 and I1 are still executing. I2 starts executing. I4 and I5 are issued. Note that I3 cannot start executing during this cycle because register x3 which is produced by I1 is still not available.

Instruction queue:

instq	{ size=0x00000000 }
-------	---------------------

## RAT



RAT	0x00ccf690 {0x00000000 <NULL>, 0x00d817b0 {instruction_name="LW" whole_instruction="LW x1,x0,10" op
▷ [0x00000000]	0x00000000 <NULL>
▷ [0x00000001]	0x00d817b0 {parameter1=0x00000000 address=0x0000000a immediate=0x0000000a ...}
▷ [0x00000002]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
▷ [0x00000003]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
▷ [0x00000004]	0x00d81868 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81700 {parameter1=0x00000000 imn
▷ [0x00000005]	0x00d81920 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d80e48 {parameter1=0x00000000 imn
▷ [0x00000006]	0x00d819d8 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81868 {parameter1=0x00000000 par
▷ [0x00000007]	0x00000000 <NULL>

Since all instructions have been issued, it can be seen that the RAT is unchanged, with the instruction queue being empty as well.

### MUL & NAND stations

NAND_stations	0x00ccf708 {0x00d819d8 {instruction_name="NAND" whole_instruction="NAND x6,x4,x2" operand1=0x00000000
MUL_stations	0x00ccf70c {0x00d81920 {instruction_name="MUL" whole_instruction="MUL x5,x2,x3" operand1=0x00000000!

Since I4 and I5 were issued during this cycle, they have been inserted into their own respective stations.

### ROB

ROB	{ size=0x00000006 }
▷ [allocator]	{ size=0x00000006 }
▷ [0]	0x00d80e48 {parameter1=0x00000000 immediate=0x00000005 p=0x00000000 <NULL> ...}
▷ [1]	0x00d81700 {parameter1=0x00000000 immediate=0x0000000a p=0x00000000 <NULL> ...}
▷ [2]	0x00d817b0 {parameter1=0x00000000 address=0x0000000a immediate=0x0000000a ...}
▷ [3]	0x00d81868 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81700 {parameter1=0x00000000 imn
▷ [4]	0x00d81920 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d80e48 {parameter1=0x00000000 imn
▷ [5]	0x00d819d8 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81868 {parameter1=0x00000000 par
▷ [Raw View]	{...}

The ROB now has all 6 instructions, since I4 and I5 were issued during this cycle.

## Cycle 5

During this cycle, I0 and I1 finish executing and go on to the writeback phase, which essentially means they will be signaled as ready for committing next cycle. I2 is still executing. I3 can now start executing, since the value from I1 can be forwarded. I4 also starts execution while I5 has to wait for I3, since it produces x4. Therefore, all stations and tables remain the same.

## Cycle 6

During this cycle, I0 and I1 are committed, i.e. are written to the register file. I2 finishes execution and is signaled to be ready. I3 and I4 are still being executed while I5 is still waiting for I3 to finish execution.

ROB

ROB	{ size=0x00000004 }
c	{ size=0x00000004 }
[allocator]	allocator
[0]	0x00d817b0 {parameter1=0x00000000 address=0x0000000a immediate=0x0000000a ...}
[1]	0x00d81868 {parameter1=0x0000000a parameter2=0x00000005 p1=0x00d81700 {parameter1=0x00000000 imr
[2]	0x00d81920 {parameter1=0x00000005 parameter2=0x0000000a p1=0x00d80e48 {parameter1=0x00000000 imr
[3]	0x00d819d8 {parameter1=0x00000000 parameter2=0x00000000 p1=0x00d81868 {parameter1=0x0000000a par

The ROB now only has 4 entries, since I0 and I1 have been committed.

LW station

LW_stations	0x00ccf6ec {0x00000000 <NULL>, 0x00000000 <NULL>}
-------------	---

The LW station has now been freed up, since I2 finished execution and went on to the writeback phase.

## Cycle 7

I2 is now ready to be committed. I3 finishes execution and passes onto the writeback stage. I4 is still executing. I5 can now start executing, since the value produced by I3 can be forwarded.

ROB

ROB	{ size=0x00000003 }
-----	---------------------

The ROB now has 3 entries only, as I2 is removed since it has committed.

ADD stations

ADD\_stations {0x00ccf6d8 {0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>}}

The ADD stations are now all free, since I3 finished execution.

RAT

RAT	0x00c8f690 {0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00d818}
[0x00000000]	0x00000000 <NULL>
[0x00000001]	0x00000000 <NULL>
[0x00000002]	0x00000000 <NULL>
[0x00000003]	0x00000000 <NULL>
[0x00000004]	0x00d81868 {parameter1=0x0000000a parameter2=0x00000005 p1=0x00d81700 {parameter1=0x00000000 imn
[0x00000005]	0x00d81920 {parameter1=0x00000005 parameter2=0x0000000a p1=0x00d80e48 {parameter1=0x00000000 imn
[0x00000006]	0x00d819d8 {parameter1=0x00000005 parameter2=0x00000005 p1=0x00d81868 {parameter1=0x0000000a par
[0x00000007]	0x00000000 <NULL>

Register x1 in the RAT has been freed up since I2 has committed.

## Cycle 8

During this cycle, I3 is ready to be committed. I5 finishes execution, since it only takes one cycle. I4 is still being executed.

ROB

ROB { size=0x00000002 }

Since I3 was committed, it was removed from the ROB.

RAT

RAT	0x00c8f690 {0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000}
[0x00000000]	0x00000000 <NULL>
[0x00000001]	0x00000000 <NULL>
[0x00000002]	0x00000000 <NULL>
[0x00000003]	0x00000000 <NULL>
[0x00000004]	0x00000000 <NULL>
[0x00000005]	0x00d81920 {parameter1=0x00000005 parameter2=0x0000000a p1=0x00d80e48 {parameter1=0x00000000 imn
[0x00000006]	0x00d819d8 {parameter1=0x00000005 parameter2=0x00000005 p1=0x00d81868 {parameter1=0x0000000a par
[0x00000007]	0x00000000 <NULL>

Register x4 was freed up, since I3 committed.

## Cycle 9

During this cycle, I5 is ready to be committed. However, since I4 is still executing and committing should be done in order, I5 remains in the ROB until I4 is committed.

### ROB & RAT

ROB	{ size=0x00000002 }
c	{ size=0x00000002 }
▸ [allocator]	allocator
▸ [0]	0x00d81920 {parameter1=0x00000005 parameter2=0x0000000a p1=0x00d80e48 {parameter1=0x00000000 imn
▸ [1]	0x00d819d8 {parameter1=0x00000005 parameter2=0x00000005 p1=0x00d81868 {parameter1=0x0000000a pari
▸ [Raw View]	{...}

RAT	0x00c8f690 {0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000
▸ [0x00000000]	0x00000000 <NULL>
▸ [0x00000001]	0x00000000 <NULL>
▸ [0x00000002]	0x00000000 <NULL>
▸ [0x00000003]	0x00000000 <NULL>
▸ [0x00000004]	0x00000000 <NULL>
▸ [0x00000005]	0x00d81920 {parameter1=0x00000005 parameter2=0x0000000a p1=0x00d80e48 {parameter1=0x00000000 imn
▸ [0x00000006]	0x00d819d8 {parameter1=0x00000005 parameter2=0x00000005 p1=0x00d81868 {parameter1=0x0000000a pari
▸ [0x00000007]	0x00000000 <NULL>

## Cycles 10 to 12

During these cycles, nothing is happening except for the execution of I4.

## Cycle 13

I4 finishes execution and goes onto the writeback stage.

## Cycle 14

I4 finally commits, which allows I5 to committed as well. Program stops execution.

ROB	{ size=0x00000000 }
-----	---------------------



RAT	0x00a7f8b0 {0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>, 0x00000000 <NULL>}
▷ [0x00000000]	0x00000000 <NULL>
▷ [0x00000001]	0x00000000 <NULL>
▷ [0x00000002]	0x00000000 <NULL>
▷ [0x00000003]	0x00000000 <NULL>
▷ [0x00000004]	0x00000000 <NULL>
▷ [0x00000005]	0x00000000 <NULL>
▷ [0x00000006]	0x00000000 <NULL>
▷ [0x00000007]	0x00000000 <NULL>

## Results

```

Program finished executing instructions
Results:
Register 0: 0
Register 1: 22
Register 2: 5
Register 3: 10
Register 4: 5
Register 5: 50
Register 6: -6
Register 7: 0
Cycles Elapsed: 14
IPC: 0.428571
Press any key to continue . . . _

```

Note: The branch miss-prediction percentage is not displayed, since there weren't any branches in the program.

## Results

We have appended our test cases along with the result of each (Register values, Cycles elapsed, IPC, etc.). In our test cases, we can see that the CPI has decreased compared to if it was our usual pipeline, but it hasn't increased enough for us to have an IPC greater than one. Our branch prediction ratio seems to be stable at 25% mispredictions, which is not a bad result.

*Note: The basic design for this project is influenced by an old design for a Simple Integer Machine done by the team-members. Some functions, such as the ones in charge of input-handling, are modified versions of functions from the old design made to fit the requirements of this project.*

# Appendix

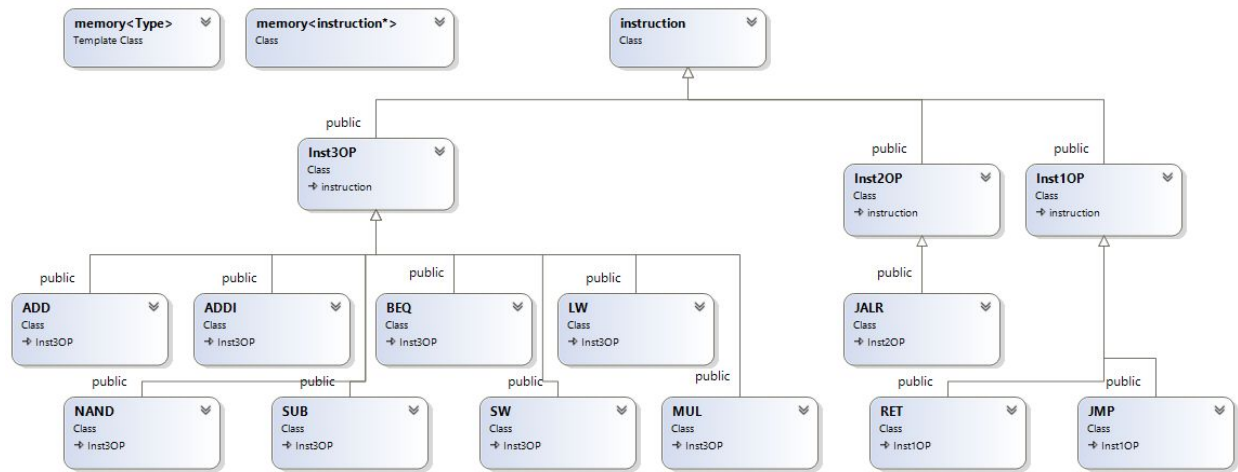


Figure 1: Class Diagram

## Test 1:

```

.text
ADDI x2,x0,5
ADDI x3,x0,10
LW x1,x0,10
SUB x4,x3,x2
MUL x5,x2,x3
NAND x6,x4,x2

.data
0x00:    20
0x01:    45
0x0A:    22
  
```

```

Results:
Register 0: 0
Register 1: 22
Register 2: 5
Register 3: 10
Register 4: 5
Register 5: 50
Register 6: -6
Register 7: 0
Cycles Elapsed: 14
IPC: 0.428571
  
```

**Test 2:**

```
.text
ADDI x1,x0,3
BEQ x7,x1,7
LW x2,x6,10
LW x3,x6,13
MUL x4,x2,x3
SW x4,x6,16
ADDI x6,x6,1
ADDI x7,x7,1
JMP -8
ADDI x5,x0,5

.data
0xA: 10
0xB: 20
0xC: 30
0xD: 2
0xE: 4
0xF: 6
```

**Test 3:**

```
.text
LW x1,x0,10
LW x2,x0,11
LW x3,x0,12
MUL x4,x2,x3
MUL x5,x1,x2
LW x6,x0,13
ADD x6,x3,x6
ADDI x7,x7,8
SUB x3,x6,x7
SW x6,x0,13

.data
0xA: 19
0xB: 2
0xC: 25
```

**Results:**

```
Register 0: 0
Register 1: 3
Register 2: 30
Register 3: 6
Register 4: 180
Register 5: 5
Register 6: 3
Register 7: 3
Cycles Elapsed: 45
IPC: 0.6
Branch Miss (%): 25%
```

**Results:**

```
Register 0: 0
Register 1: 19
Register 2: 2
Register 3: 225
Register 4: 50
Register 5: 38
Register 6: 233
Register 7: 8
Cycles Elapsed: 20
IPC: 0.5
```

0xD: 208

#### Test 4:

```
.text
LW x1,x0,20
ADDI x2,x0,21
ADD x3,x2,x1
ADDI x4,x0,0
BEQ x4,x1,7
LW x5,x2,0
MUL x5,x5,x1
SW x5,x3,0
ADDI x2,x2,1
ADDI x3,x3,1
ADDI x4,x4,1
JMP -8
ADD x0,x0,x0
```

```
.data
0x14:      3
0x15:     12
0x16:     -5
0x17:      7
```

#### Test 5:

```
.text
ADDI x1,x0,8
ADDI x2,x0,4
ADDI x3,x0,-5
ADD x4,x1,x2
MUL x0,x2,x2
SW x4,x0,11
LW x5,x0,10
LW x6,x0,11
SUB x7,x5,x1
NAND x4,x1,x2
```

```
.data
```

#### Results:

```
Register 0: 0
Register 1: 3
Register 2: 24
Register 3: 27
Register 4: 3
Register 5: 21
Register 6: 0
Register 7: 0
Cycles Elapsed: 49
IPC: 0.612245
Branch Miss (%): 25%
```

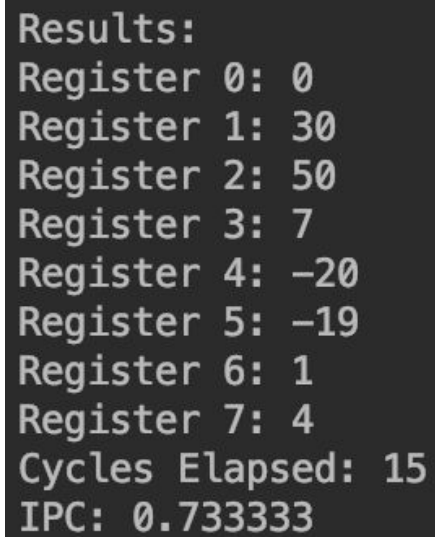
#### Results:

```
Register 0: 0
Register 1: 8
Register 2: 4
Register 3: -5
Register 4: -1
Register 5: 0
Register 6: 0
Register 7: -8
Cycles Elapsed: 20
IPC: 0.5
```

0x0: 20  
0x1: 45  
0xA: 22

**Test 6:**

```
.text  
ADDI x1,x0,22  
ADDI x2,x0,10  
ADDI x3,x0,7  
JALR x7,x3  
SUB x4,x1,x2  
NAND x5,x1,x2  
JMP 3  
ADDI x1,x0,30  
ADDI x2,x0,50  
RET x7  
ADDI x6,x0,1
```

A dark-themed rectangular box containing the results of the assembly test. The text is white and lists the final values of registers 0 through 7, the total cycles elapsed, and the instructions per cycle (IPC) value.

**Results:**  
Register 0: 0  
Register 1: 30  
Register 2: 50  
Register 3: 7  
Register 4: -20  
Register 5: -19  
Register 6: 1  
Register 7: 4  
Cycles Elapsed: 15  
IPC: 0.733333