

TRABAJO PRACTICO DE ARQUITECTURA DE COMPUTADORAS

ALGORITMO DE TOMASULO

Profesor: Ing. Orlando Micolini

Asignatura: Arquitectura de Computadoras

Integrantes:

Aguilar, Mauricio

Tarazi, Pedro Esequiel

22 de octubre de 2018

Contenido

INTRODUCCIÓN	3
DESARROLLO	4
DIAGRAMAS	5
IMPLEMENTACIÓN	10
CASOS DE PRUEBA.....	12
CONCLUSIÓN.....	17
BIBLIOGRAFÍA	18
ANEXO.....	19
PROGRAMA 1	19
PROGRAMA 2	19
PROGRAMA 3	20
PROGRAMA 4	20

INTRODUCCIÓN

El Algoritmo de Tomasulo es un algoritmo de planificación dinámica, diseñado para permitir a un procesador ejecutar instrucciones fuera de orden. Este algoritmo resuelve los riesgos WAW y WAR realizando un renombrado, lo que permite que las instrucciones se lancen. Además utiliza un bus de datos común en el que los valores calculados son enviados a todas las estaciones de reserva que lo necesiten. Esto permite mejorar la ejecución paralela de instrucciones en situaciones en las que otros algoritmos fallarían y provocarían una detención.

En la actualidad, gran parte de los procesadores hacen uso de variaciones de este algoritmo para la planificación dinámica de instrucciones.

En este trabajo se realizó la implementación del Algoritmo de Tomasulo en lenguaje de programación Java a fin de entender su funcionamiento. El algoritmo debe incluir un buffer de reordenamiento, conocido como ROB (Re-Order Buffer). Esto permitirá que las instrucciones ingresen en orden, se ejecuten fuera de orden, y salgan del sistema nuevamente en orden.

Se implementaron las siguientes partes:

- Unidad Funcional Sumador
- Estaciones de Reserva para el Sumador
- Unidad Funcional Multiplicador
- Estaciones de Reserva para el Multiplicado
- Unidad Funcional Cargador (Loader)
- Estaciones de Reserva para el Cargador
- Common Data Bus
- Re-Order Buffer
- Banco de Registros
- Banco de Memoria

DESARROLLO

El Algoritmo de Tomasulo con buffer de reordenamiento consta de las siguientes etapas:

1. Issue:

- Obtiene una instrucción de la cola.
- Despacha la instrucción si hay libre una estación de reserva y una entrada en el ROB.
- Envía los operandos a la estación de reserva si ellos están disponibles en los registros o en el ROB.
- El número de entrada del ROB utilizado es también enviado a la estación de reserva, el cual será usado cuando se necesite indicar el tag en el CDB.
- Si todas las estaciones de reserva y/o las entradas del ROB están llenas, entonces el despacho de la instrucción es detenido hasta que ambos tengan entradas disponibles.

2. Execute:

- Si uno o más de los operandos no está disponible, las estaciones de reservas monitorean el CDB en busca de valores que le puedan servir a esa estación.
- Cuando ambos operandos están disponibles en una estación de reserva, se ejecuta la operación.
- Las instrucciones pueden tener multiples ciclos de reloj. Por ejemplo:
 - ADD: 3 clocks
 - MUL: 5 clocks
 - LOAD: 2 clocks

3. Write Result:

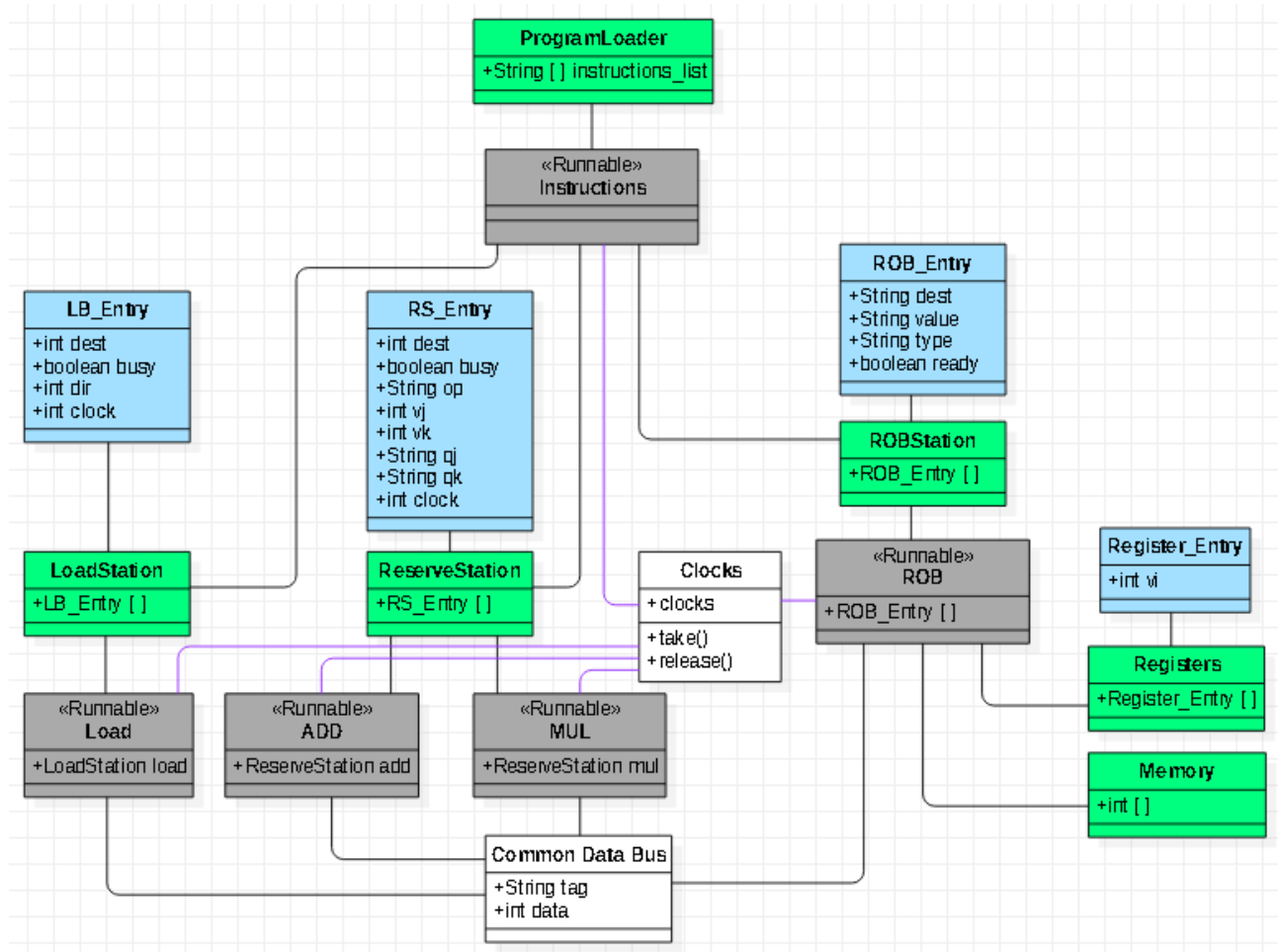
- Cuando el resultado está disponible, escribe en el CDB para el ROB.
- El ROB lee el CDB y determina a que entrada pertenece ese dato.
- Se marca esa entrada del ROB como Ready, para indicar que ya puede escribirse en memoria o en los registros.

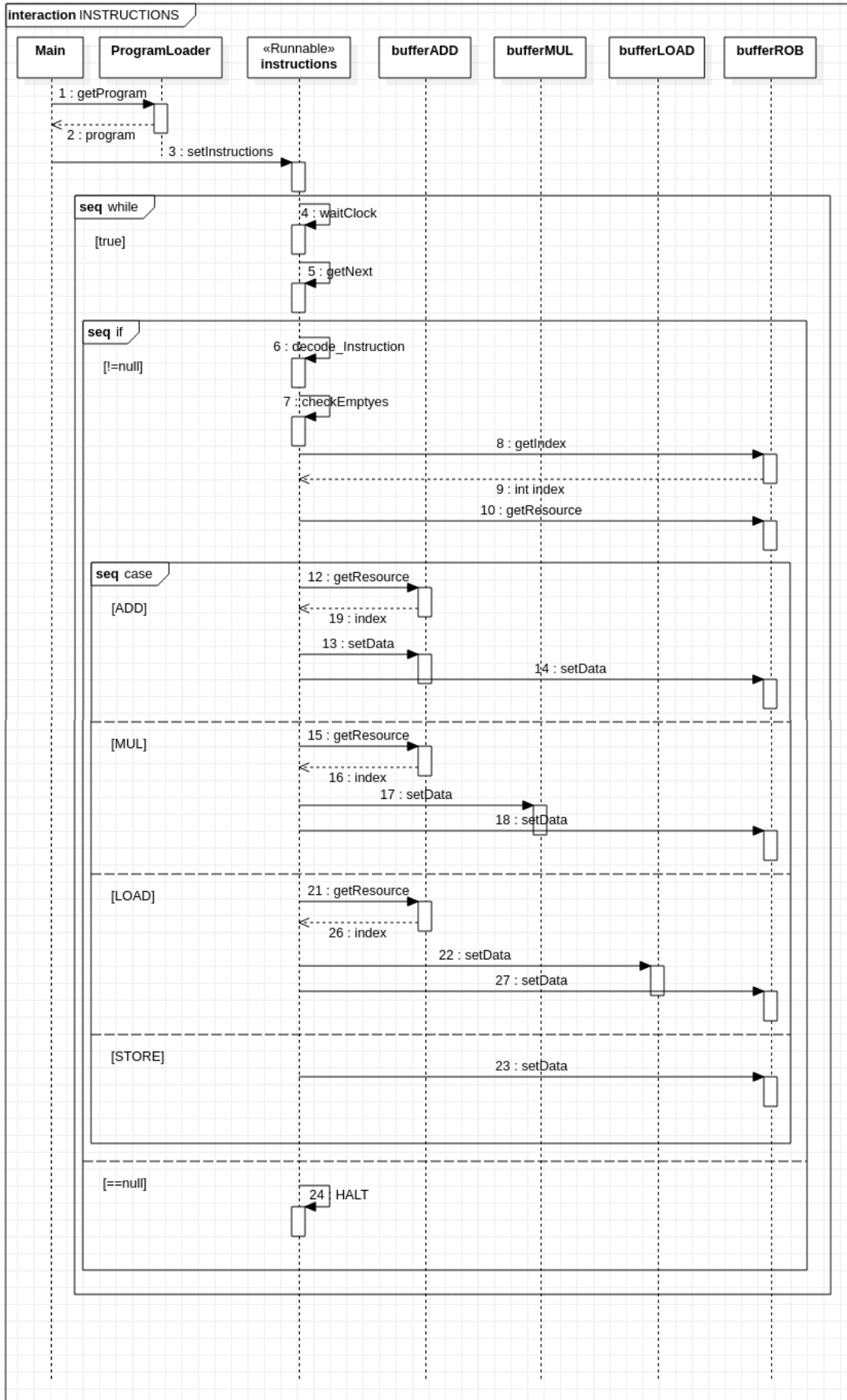
4. Commit:

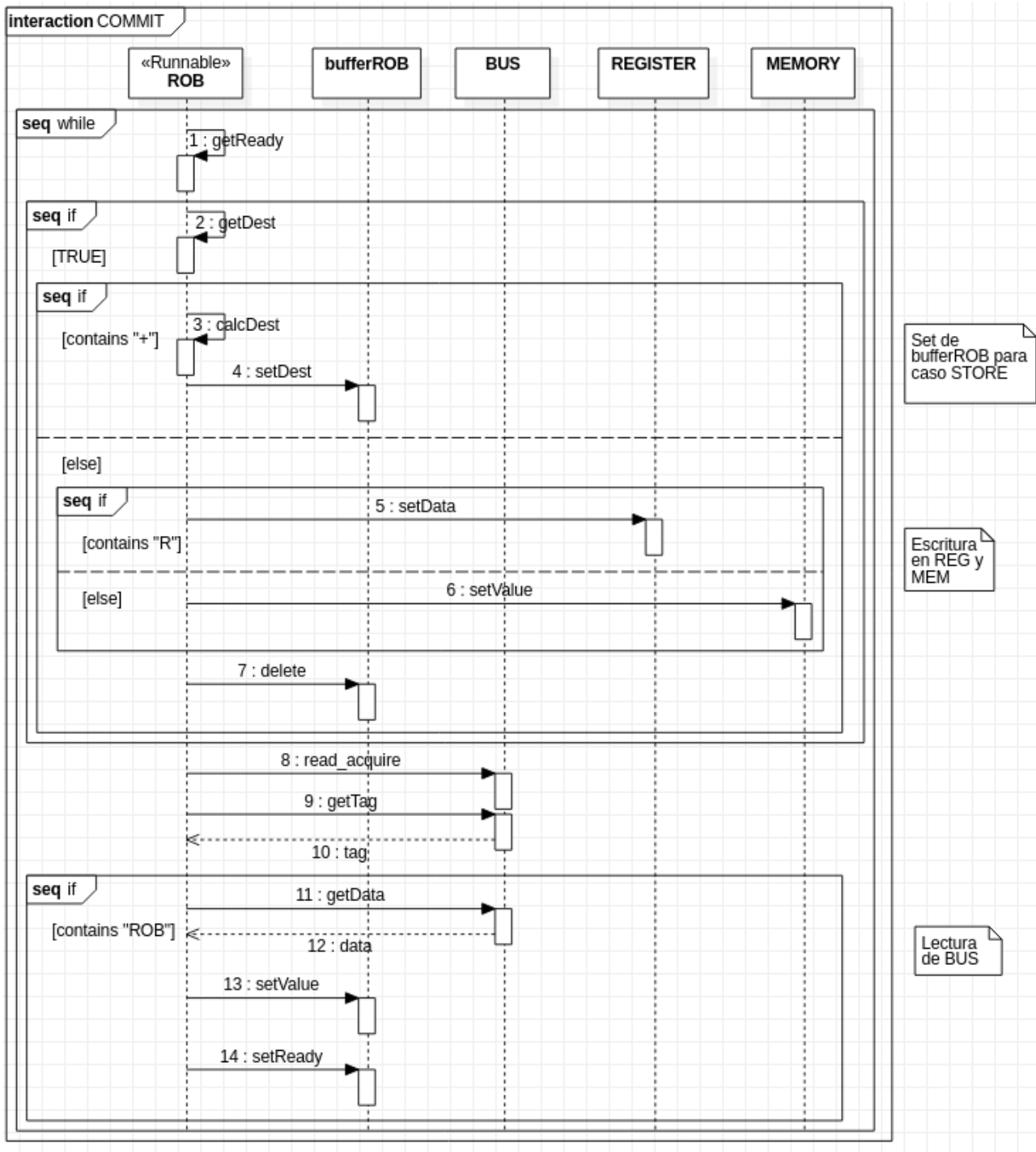
- Cuando la instrucción en la cabeza del ROB tiene su resultado presente (Ready), entonces esta instrucción esta lista para escribirse en memoria o registros.
- Se escribe su resultado, se limpia la entrada y la cabecera avanza un lugar.

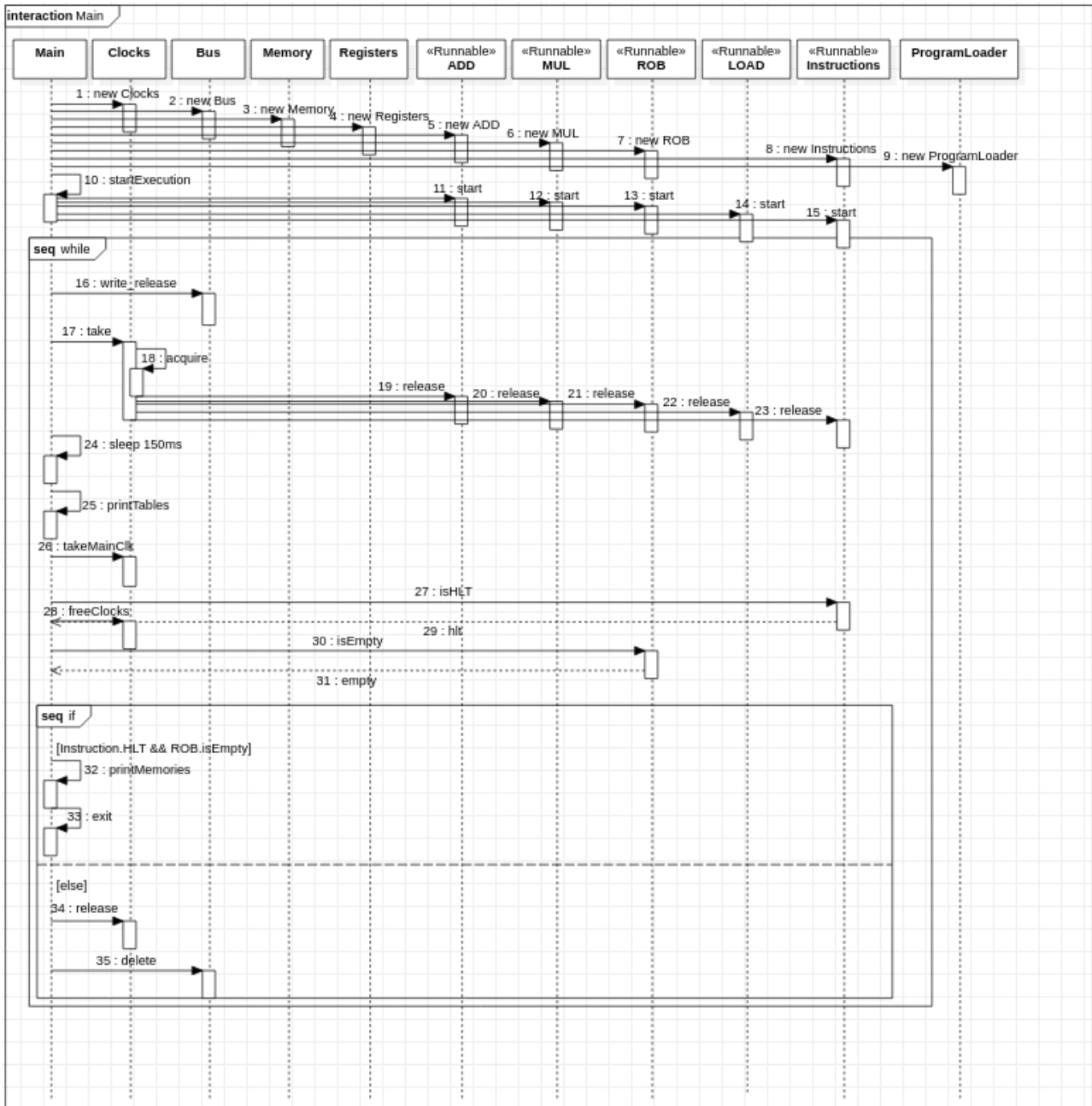
DIAGRAMAS

Luego de estudiar su funcionamiento, se procedió a elaborar un diagrama de clases y un diagrama de secuencias iniciales, como bosquejo inicial a modo de entender cuál es la mejor manera de implementarlo, y descubrir cuestiones que no aparecían en primera instancia.









IMPLEMENTACIÓN

Como se muestra en el Diagrama de Clases, la implementación consta de seis clases que implementan la interface Runnable, y una clase que cumple la función de recurso compartido.

En primer lugar, tenemos la clase *Clocks*, la cual tiene un semáforo denominado “*clk*” cuya función principal es sincronizar la ejecución del resto de los hilos mediante el pedido o liberación del semáforo que corresponde a cada uno de ellos. Además, se encarga de liberar el *Common Data Bus* para que los hilos puedan escribir en él, y de borrar los datos que contiene el mismo una vez que todos lo leyeron. Esto ultimo se realiza una vez que los hilos liberaron el semáforo de lectura de bus correspondiente, el cual será explicado más adelante.

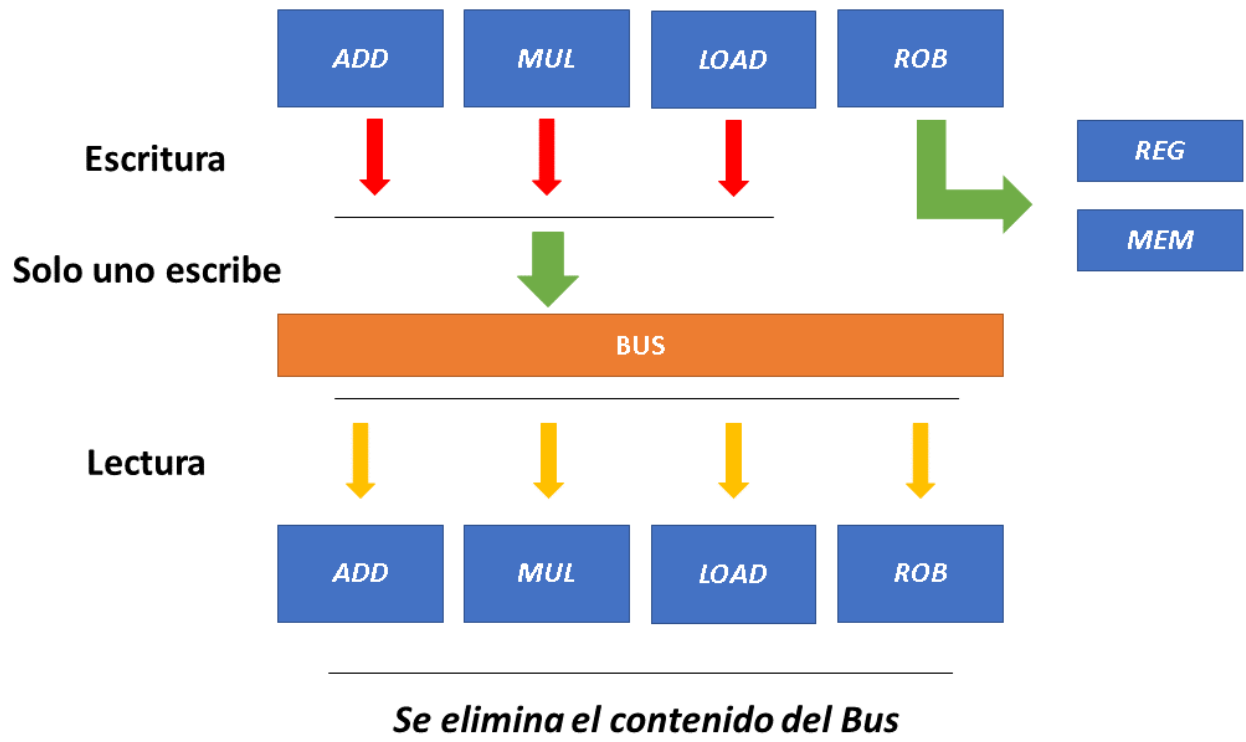
Luego tenemos la clase *Instructions*, la cual es un hilo cuya función es leer las instrucciones desde un archivo de texto, y colocarlas en la Estación de Reserva (ER) que corresponda y en el Buffer de Reordenamiento (ROB). Una vez que todas las instrucciones fueron asignadas, su tarea ha sido completada.

Los hilos ADD, MUL, y LOAD tienen un funcionamiento similar. Estos corresponden a las Unidades Funcionales (UF). Lo primero que hacen estos hilos es un proceso de escritura, el cual puede ser completado (escribe en el bus) o no (no escribe en el bus). Este proceso consiste en recorrer las ER de su buffer y verificar si hay instrucciones cargadas en las mismas, chequear que contengan todos los operandos, y que cumplan con los ciclos de clocks de ejecución. Una vez que estas condiciones se cumplen, la UF intenta tomar el bus para escribir en él: si logra tomarlo, se realiza la operación, se escriben en él, y se borra la ER, de lo contrario, intentará en el siguiente ciclo de clock. Finalizado este proceso, la UF “avisa” liberando el semáforo **común de lectura** que su proceso de escritura finalizó. Cada UF intentará tomar el semáforo de lectura de bus correspondiente a la misma, pero este estará bloqueado hasta que las tres UF completen el proceso de escritura. Luego, se procede con la lectura de bus, la cual consiste en leerlo y comparar lo que se obtuvo con los operandos de las ER. Si alguno coincide, se reemplaza. Finalizado esto, cada UF libera su semáforo de lectura para indicar que se leyó el bus y que los datos de éste ultimo pueden ser eliminados.

El ROB tiene un funcionamiento similar a las UF con la diferencia que la escritura la realiza sobre los Registros y Memoria, y no sobre el bus. Terminado este proceso, intentará, al igual que las UF, continuar con el proceso de lectura del bus, el cual es el mismo que se comentó anteriormente. Finalizado esto, libera su semáforo de lectura permitiendo así eliminar los datos del bus.

La clase Bus esta compuesta de un semáforo de escritura (puede escribir en el bus solo una estación por ciclo de clock), cuatro semáforos de lectura (uno para cada UF y uno para el ROB), y un semáforo para avisar que se pueden borrar los datos.

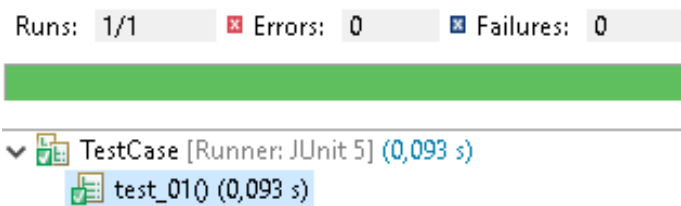
Para terminar, los datos que contiene el bus se eliminaran una vez que TODAS las unidades funcionales hayan terminado su proceso de lectura. Luego de esto, comenzará el siguiente ciclo de clock.



CASOS DE PRUEBA

Se realizaron tres test de comprobación del funcionamiento del Re-Order Buffer (ROB):

ID_Test	Test_01
Nombre del Test	Carga de instrucción en ROB
Requerimiento	La instrucción debe cargarse en el ROB siempre y cuando haya lugar disponible en él y en la estación de reserva (ER).
Precondiciones	Se debe crear un archivo el cual contiene las instrucciones del programa a ejecutar. El mismo debe llamarse “program#.txt”. El archivo debe contener una instrucción por línea. Las instrucciones son del tipo: ADD R1 R2 R3
Secuencia	Ciclo 1: “Instructions” lee la primera instrucción del buffer. Luego verifica si hay lugar en el ROB y en la ER correspondiente. Si hay lugar, se carga la instrucción.
Resultado Esperado	La instrucción debe estar cargada en el buffer de ROB y en el buffer de la ER correspondiente.
Resultado Obtenido	La instrucción se cargó en el ROB y en la ER
Estado del TEST	EXITOSO



ID_Test	Test_02
Nombre del Test	Bloqueo de carga en ROB por buffer lleno
Requerimiento	Si no hay lugar en el buffer del ROB, se debe detener la carga de instrucciones hasta que se desocupe un lugar en él.
Precondiciones	Se debe crear un archivo el cual contiene las instrucciones del programa a ejecutar. El mismo debe llamarse "program#.txt". El archivo debe contener una instrucción por línea. Las instrucciones son del tipo: ADD R1 R2 R3 El ROB debe tener tamaño 1.
Secuencia	Ciclo 1. "Instructions" lee la primera instrucción del buffer. Es un ADD. Verifica si hay lugar en el ROB y en ADD. Tiene lugar, carga la instrucción y toma la siguiente. Ciclo 2: "Instructions" lee la segunda instrucción del buffer. Es un LD. Como ROB no tiene lugar, detiene la carga hasta que se libere. Ciclos 2,3 y 4: ADD se ejecuta. Ciclo 5: Luego de que la primera instrucción se escribió en el registro correspondiente, se libera el lugar en ROB. Ciclo 6: Se carga la instrucción LD en ROB y en LOAD.
Resultado Esperado	La carga de instrucciones en debe bloquearse a la espera de lugar disponible.
Resultado Obtenido	La carga de instrucciones se bloqueó.
Estado del TEST	Exitoso

Runs: 1/1 Errors: 0 Failures: 0

▼ TestCase [Runner: JUnit 5] (0,653 s)

 test_02() (0,653 s)

ID_Test	Test_04
Nombre del Test	ROB escribe en orden
Requerimiento	ROB debe escribir las instrucciones en Registro y/o Memoria en el orden en el que fueron despachadas desde la unidad Instructions.
Precondiciones	Se debe crear un archivo el cual contiene las instrucciones del programa a ejecutar. El mismo debe llamarse "program#.txt". El archivo debe contener una instrucción por línea. Las instrucciones son del tipo: ADD R1 R2 R3
Secuencia	<p>Se lee una instrucción ADD. Si hay lugar, se carga en ROB y en la ER correspondiente.</p> <p>Se ejecuta el programa y se comprueba si se va a remover una instrucción del ROB, con el campo Ready.</p> <p>Si la instrucción en el ROB se va a commitear, se comprueba que coincida con el orden en el que fueron leídas en la unidad Instructions.</p>
Resultado Esperado	Las instrucciones finalicen en orden.
Resultado Obtenido	Las instrucciones finalizaron en orden.
Estado del TEST	Exitoso

Finished after 1,579 seconds


Runs: 1/1 ✖ Errors: 0 ❌ Failures: 0


▼ 📄 TestCase [Runner: JUnit 5] (1,449 s)


📄 test_04() (1,449 s)

ID_Test	Test_05
Nombre del Test	Resultados correctos
Requerimiento	Los programas deben finalizar con resultados correctos.
Precondiciones	Se debe crear un archivo el cual contiene las instrucciones del programa a ejecutar. El mismo debe llamarse “program#.txt”. El archivo debe contener una instrucción por línea. Las instrucciones son del tipo: ADD R1 R2 R3
Secuencia	Se carga un programa completo y se ejecuta hasta finalizar.
Resultado Esperado	Los registros y memorias deben tener resultado correctos
Resultado Obtenido	Los registros y memorias tienen resultados correctos.
Estado del TEST	Exitoso

Runs: 1/1 ✖ Errors: 0 ❌ Failures: 0



▼  TestCase [Runner: JUnit 5] (5,293 s)

 test_05() (5,293 s)

CONCLUSIÓN

Se desarrolló el Algoritmo de Tomasulo con Buffer de Reordenamiento. La ejecución fuera de orden es una de sus principales ventajas, ya que permite que el procesador pueda seguir ejecutando aquellas instrucciones que no sean dependientes de sus antecesoras y que no se encuentren en la ventana de instrucciones, es decir, actualmente en ejecución en el procesador. Entonces puede intercalar instrucciones y aprovechar casi al máximo al procesador. El procesamiento de instrucciones solo se detiene si no hay una estación de reserva o una entrada en el ROB disponible para la próxima instrucción. Por lo tanto haciendo ajustes en el tamaño del ROB podemos ver que con una cantidad de entrada igual a la suma de las cantidades de estaciones de reserva, el sistema funciona sin detenciones.

En cuanto al Common Data Bus, se ha implementado con un semáforo, y por tanto su política es la siguiente: Cualquiera de las unidades funciones que necesite escribir en el CDB, compite por el semáforo, y aquella que lo gane podrá escribir. Para evitar que haya inanición sobre las estaciones de reserva de una misma unidad funcional, se implementó la ejecución de modo de darle prioridad a estas en un orden Round-Robin. No se ha implementado una política para evitar la inanición de una unidad funcional, ya que solo son tres, sus instrucciones se ejecutan en distintas cantidades de clocks y el sistema de renombrado hace que haya un orden implícito entre ciertas instrucciones por lo cual sería lógico pensar que si solo hay tres estaciones de reserva en las unidades de suma y multiplicación, estas quedarán ociosas rápidamente luego de algunas escrituras en el buffer.

Mediante la realización de este trabajo logramos comprender el funcionamiento de un procesador con el Algoritmo de Tomasulo, así como sus ventajas y desventajas.

BIBLIOGRAFÍA

- Computer Architecture: A Quantitative Approach 5th Edition by John L. Hennessy (Author), David A. Patterson (Author)
- Tema 3 – ILP, Planificación Dinamica, Prediccion de Saltos, Especulacion - Arquitectura de Computadoras – Orlando Micolini

ANEXO

PROGRAMA 1

ADD R0 R1 R2

LD R1 1 R2

ADD R2 R1 R4

ST 1 R4 R1

MUL R4 R6 R2

ADD R3 R4 R5

Resultado:

R0 = 3 M0 = 0

R1 = 3 M1 = 1

R2 = 7 M2 = 2

R3 = 47 M3 = 3

R4 = 42 M4 = 4

R5 = 5 M5 = 3

R6 = 6 M6 = 6

R7 = 7 M7 = 7

R8 = 8 M8 = 8

PROGRAMA 2

ADD R0 R1 R2

ADD R1 R2 R3

ADD R2 R3 R4

ADD R4 R5 R6

ADD R5 R6 R7

Resultado:

R0 = 3 M0 = 0

R1 = 5 M1 = 1

R2 = 7 M2 = 2

R3 = 3 M3 = 3

R4 = 11 M4 = 4

R5 = 13 M5 = 5

R6 = 6 M6 = 6

R7 = 7 M7 = 7

R8 = 8 M8 = 8

PROGRAMA 3

ADD R0 R1 R2

LD R1 1 R0

ADD R2 R1 R4

ST 1 R4 R1

MUL R4 R6 R2

ADD R3 R4 R5

Resultado:

R0 = 3 M0 = 0

R1 = 4 M1 = 1

R2 = 8 M2 = 2

R3 = 53 M3 = 3

R4 = 48 M4 = 4

R5 = 5 M5 = 4

R6 = 6 M6 = 6

R7 = 7 M7 = 7

R8 = 8 M8 = 8

PROGRAMA 4

LD R1 1 R4

LD R2 0 R3

ADD R4 R0 R2

LD R0 0 R1

ADD R5 R0 R2

MUL R5 R1 R2

MUL R6 R2 R3

MUL R7 R3 R4

MUL R8 R4 R5

ST 0 R0 R7

ST 0 R2 R8

Resultado:

R0 = 5 R6 = 9 M0 = 0 M6 = 6

R1 = 5 R7 = 9 M1 = 1 M7 = 7

R2 = 3 R8 = 45 M2 = 2 M8 = 8

R3 = 3 M3 = 45

R4 = 3 M4 = 4

R5 = 15 M5 = 9